Europäisches Patentamt
European Patent Office
Office européen des brevets

Bescheinigung Certificate Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. Patent application No. Demande de brevet n°

03015015.5

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

R C van Dijk

Anmeldung Nr:
Application no.: 03015015.5
Demande no:

Anmeldetag:
Date of filing: 02.07.03
Date de dépôt:

Anmelder/Applicant(s)/Demandeur(s):

PACT XPP Technologies AG
Muthmannstrasse 1
80939 München
ALLEMAGNE

Bezeichnung der Erfindung/Title of the invention/Titre de l'invention:
(Falls die Bezeichnung der Erfindung nicht angegeben ist, siehe Beschreibung.
If no title is shown please refer to the description.
Si aucun titre n'est indiqué se referer à la description.)

Method for operating an array of reconfigurable elements processing data

In Anspruch genommene Priorität(en) / Priority(ies) claimed /Priorité(s) revendiquée(s)
Staat/Tag/Aktenzeichen/State/Date/File no./Pays/Date/Numéro de dépôt:

Internationale Patentklassifikation/International Patent Classification/ Classification internationale des brevets:

G06F15/78

Am Anmeldetag benannte Vertragstaaten/Contracting states designated at date of filing/Etats contractants désignées lors du dépôt:

AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IT LU MC NL
PT RO SE SI SK TR LI

The present invention relates to methods of operating reconfigurable arrays of data processing elements.

When using sucharrays, it is desired to optimise the way the array is coupled to other units, e.g. to a processor if used as a coprocessor and/or to optimise the way in which the array is configured.

The present invention aims at providing improvments over the prior art.

It is to be noted that the disclosure of the present invention does comprise two major parts in its description that both refer to ways of allowing for an optimum use of the array and hence are closely related to each other.

It is also to be noted that the shorter of the two major parts does comprise a plurality of figures that the text relates to however without always giving an exact, precise and correct reference. Yet any deviations from correct referencing will be obvious to the average skilled person.

Table of Contents

1 Executive Summary

The study is concerned with three objectives:

1. Proposal of a hardware framework, which enables an efficient integration of the PACT XPP core into a standard RISC processor architecture.

2. Proposal of a compiler for the coupled RISC+XPP hardware. This compiler decides automatically which part of a source code is executed on the RISC processor and which part is executed on the PACT XPP.

3. Presentation of a number of case studies demonstrating which results may be achieved by using the proposed C Compiler in cooperation with the proposed hardware framework.

The proposed hardware framework accelerates the XPP core in two respects. First, data throughput is increased by raising the XPP's internal operating frequency into the range of the RISC's frequency. This, however, means that the XPP runs into the same pit like all high frequency processors - memory accesses become very slow compared to processor internal computations. This is why the use of a cache is proposed. It eases the memory access problem for a large range of algorithms, which are well suited for an execution on the XPP. The cache as second throughput increasing feature requires a controller. Hence a programmable cache controller is introduced, which manages the cache contents and feeds the XPP core. It decouples the XPP core computations from the data transfer so that, for instance, data preload to a specific cache sector takes place while the XPP is operating on data located in a different cache sector.

Another problem emerging with a coupled RISC+XPP hardware is concerned with the RISC's multitasking concept. It becomes necessary to interrupt computations on the XPP in order to perform a task switch. Multitasking is supported by the proposed compiler, as well as by the proposed hardware. First, each XPP configuration is considered as an uninterruptible entity. This means that the compiler, which generates the configurations, takes care that the execution time of any configuration does not exceed a predefined time slice. Second, the cache controller is concerned with the saving and restoring of the XPP's state after an interrupt. The proposed cache concept minimizes the memory traffic for interrupt handling and frequently even allows avoiding memory accesses at all.

Finally, the proposed cache concept is based on a simple IRAM cell structure allowing for an easy scalability of the hardware - extending the XPP cache size, for instance, requires not much more than the duplication of IRAM cells.

The study proposes a compiler for a RISC+XPP system. The objective of the compiler is that real-world applications, which are written in the C language, can be compiled for a RISC+XPP system. The compiler removes the necessity of developing NML code for the XPP by hand. It is possible, instead, to implement algorithms in the C language or to directly use existing C applications without much adaptation to the XPP system. The proposed compiler includes three major components to perform the compilation process for the XPP:

1. partitioning of the C source code into RISC and XPP parts,

2. transformations to optimize the code for the XPP and

3. generating NML code.

Finally the generated NML code is placed and routed for the XPP.

The partitioning component of the compiler decides which parts of an application code can be executed on the XPP and which parts are executed on the RISC. Typical candidates for becoming XPP code are loops with a large number of iterations whose loop bodies are dominated by arithmetic operations. The remaining source code - including the data transfer code - is compiled for the RISC.

The proposed compiler transforms the XPP code such that it is optimized for NML code generation. The transformations included in the compiler comprise a large number of loop transformations as well as general code transformations. Together with data and code analysis the compiler restructures the code so that it fits into the XPP array and that the final performance exceeds the pure RISC performance. Finally the compiler generates NML code from the transformed program. The whole compilation process is controlled by an optimization driver which selects the optimal order of transformations based on the source code.

The case studies build a major aspect of the study. The selection of the examples is conducted by the guiding principle that each example stands for a set of typical real-world applications. For each example the study demonstrates the work of the proposed compiler. First the code is partitioned. The code transformations, which are done by the compiler, are shown and explained. Some examples require minor source code transformations which must be performed by hand. The study argues that these transformations are either too expensive, or too specific to make sense to be included in the proposed compiler. Dataflow graphs of the transformed codes are constructed for each example, which are used by the compiler to generate the NML code. In addition the XPP resource usages are shown.

The case studies demonstrate that a compiler containing the proposed transformations can generate efficient code from numerical applications for the XPP. This is possible because the compiler relies on the features of the suggested hardware, like the cache controller.

2 Hardware

2.1 Design Parameter Changes

Since the XPP core shall be integrated as a functional unit into a standard RISC core, some system parameters have to be reconsidered:

2.1.1 Pipelining / Concurrency / Synchronicity

RISC instructions of totally different type (Ld/St, ALU, Mul/Div/MAC, FPALU, FPMul...) are executed in separate specialized functional units to increase the fraction of silicon that is busy on average. Such functional unit separation has led to superscalar RISC designs, that exploit higher levels of parallelism.

Each functional unit of a RISC core is highly pipelined to improve throughput. Pipelining overlaps the execution of several instructions by splitting them into unrelated phases, which are executed in different stages of the pipeline. Thus different stages of consecutive instructions can be executed in parallel with each stage taking much less time to execute. This allows higher core frequencies.

Since the pipelines of all functional units are approximately subdivided into sub-operations of the same size (execution time), these functional units / pipelines execute in a highly synchronous manner with complex floating point pipelines being the exception.

Since the XPP core uses data flow computation, it is pipelined by design. However, a single configuration usually implements a loop of the application, so the configuration remains active for many cycles, unlike the instructions in every other functional unit, which typically execute for one or two cycles at most. Therefore it is still worthwhile to consider the separation of several phases (e.g.: Ld / Ex / Store) of an XPP configuration (= XPP instruction) into several functional units to improve concurrency via pipelining on this coarser scale. This also improves throughput and response time in conjunction with multi tasking operations and implementations of simultaneous multithreading (SMT).

The multi cycle execution time also forbids a strongly synchronous execution scheme and rather leads to an asynchronous scheme, like for e.g. floating point square root units. This in turn necessitates the existence of explicit synchronization instructions.

2.1.2 Core frequency / Memory Hierarchy

As a functional unit, the XPP's operating frequency will either be half of the core frequency or equal to the core frequency of the RISC. Almost every RISC core currently on the market exceeds its memory bus frequency with its core frequency by a larger factor. Therefore caches are employed, forming what is commonly called the memory hierarchy: Each layer of cache is larger but slower than its predecessors.

This memory hierarchy does not help to speed up computations which shuffle large amounts of data, with little or no data reuse. These computations are called "bounded by memory bandwidth". However other types of computations with more data locality (another name for data reuse) gain performance as long as they fit into one of the upper layers of the memory hierarchy. This is the class of applications that gain the highest speedups when a memory hierarchy is introduced.

Classical vectorization can be used to transform memory-bounded algorithms, with a data set too big to fit into the upper layers of the memory hierarchy. Rewriting the code to reuse smaller data sets sooner exposes memory reuse on a smaller scale. As the new data set size is chosen to fit into the caches of the memory hierarchy, the algorithm is not memory bounded any more, yielding significant speed-ups.

2.1.3 Software / Multitasking Operating Systems

As the XPP is introduced into a RISC core, the changed environment – higher frequency and the memory hierarchy – not only necessitate reconsideration of hardware design parameters, but also a reevaluation of the software environment.

Memory Hierarchy

The introduction of a memory hierarchy enhances the set of applications that can be implemented efficiently. So far the XPP has mostly been used for algorithms that read their data set in a linear manner, applying some calculations in a pipelined fashion and writing the data back to memory. As long as all of the computation fits into the XPP array, these algorithms are memory bounded. Typical applications are filtering and audio signal processing in general.

But there is another set of algorithms, that have even higher computational complexity and higher memory bandwidth requirements. Examples are picture and video processing, where a second and third dimension of data coherence opens up. This coherence is e.g. exploited by picture and video compression algorithms, that scan pictures in both dimensions to find similarities, even searching consecutive pictures of a video stream for analogies. Naturally these algorithms have a much higher algorithmic complexity as well as higher memory requirements. Yet they are data local, either by design or they can be transformed to be, thus efficiently exploiting the memory hierarchy and the higher clock frequencies of processors with memory hierarchies.

Multi Tasking

The introduction into a standard RISC core makes it necessary to understand and support the needs of a multitasking operating system, as standard RISC processors are usually operated in multitasking environments. With multitasking, the operating system switches the executed application on a regular basis, thus simulating concurrent execution of several applications (tasks). To switch tasks, the operating system has to save the state (context i.e. the contents of all registers ...) of the running task and then reload the state of another task. Hence it is necessary to determine what the state of the processor is, and to keep it as small as possible to allow efficient context switches.

Modern microprocessors gain their performance from multiple specialized and deeply pipelined functional units and high memory hierarchies, enabling high core frequencies. But high memory hierarchies mean that there is a high penalty for cache misses due to the difference between core and memory frequency, since many core cycles pass until the values are finally available from memory. Deep pipelines incur pipeline stalls due to data dependencies as well as branch penalties for mispredicted conditional branches. Specialized functional units like floating point units idle for integer-only programs. For these reasons, average functional unit utilization is much too low.

The newest development with RISC processors, Simultaneous MultiThreading (SMT), adds hardware support for a finer granularity (instruction / functional unit level) switching of tasks, exposing more than one independent instruction stream to be executed. Thus, whenever one instruction stream stalls or doesn't utilize all functional units, the other one can jump in. This improves functional unit utilization for today's processors.

With SMT, the task (process) switching is done in hardware, so the processor state has to be duplicated in hardware. So again it is most efficient to keep the state as small as possible. For the combination of the PACT XPP and a standard RISC processor, SMT is very beneficial, since the XPP configurations execute longer than the average RISC instruction. Thus another task can utilize the other functional units, while a configuration is running. On the other side, not every task will utilize the XPP, so while one such non-XPP task is running, another one will be able to use the XPP core.

2.2 Communication Between the RISC Core and the XPP Core.

2.2.1 Streaming

Since streaming can only support (number_of_IO_ports * width_of_IO_port) bits per cycle, it is only well suited for small XPP arrays with heavily pipelined configurations that feature few inputs and outputs. As the pipelines take a long time to fill and empty while the running time of a configuration is limited (as will be described under "context switches"), this type of communication does not scale well to bigger arrays and array frequencies near the RISC core frequency.

- Streaming from the RISC core

  In this setup, the RISC supplies the array with the streaming data. Since the RISC core has to execute several instructions to compute addresses and load an item from memory, this setup is only suited, if the XPP core is reading data with a frequency much lower than the RISC core frequency.

- Streaming via DMA

  In this mode the RISC core only initializes a DMA channel which then supplies the data items to the streaming port of the XPP core.

2.2.2 Shared Memory (Main Memory)

In this configuration the XPP array configuration uses a number of PAEs to generate an address that is used to access main memory through the IO ports. As the number of IO ports is very limited this approach suffers from the same limitations as the previous one, although for larger arrays the impact of using PAEs for address generation is lessening. However this approach is still useful for loading values from very sparse arrays.

2.2.3 Shared Memory (IRAM)

This data access mechanism uses the IRAM array elements to store data for local computations. The IRAMs can either be viewed as vector registers or as local copies of main memory.

There are several ways to fill the IRAMs with data.

- The IRAMs can be loaded in advance by separate "load" configurations using streaming.

  This corresponds to the usage as vector registers. As explicated above, this will limit the performance of the XPP array, especially as the IRAMs will always be part of the externally visible state and hence must be saved and restored on context switches.

- The IRAMs can be loaded in advance by separate "load" instructions.

  This can be viewed as hard coded load configuration and reduces configuration reloads. Additionally, the special load instructions may use a wider interface to the memory hierarchy. This corresponds to the usage as vector registers.

- The IRAMs can be loaded by a "burst preload from memory" instruction of the cache controller.

- The best mode however is a combination of the previous solutions with the extension of a cache:

  A preload instruction maps a specific memory area defined by starting address and size to IRAMx. This triggers a (delayed, low priority) burst load from the memory hierarchy (cache). After all IRAMs are mapped, the next configuration can be activated. The activation incurs a wait until all burst loads are completed. However, if the preload instructions are issued long enough in advance and no interrupt or task switch destroys cache locality, the wait will not consume any time.

  To specify a memory block as output IRAM, a "preload clean" instruction is used, which avoids loading data from memory.

  A synchronization instruction is needed to make sure that the content of a specific memory area, which is cached in IRAM, is written back to the memory hierarchy. This can be done globally (full write back), or selectively by specifying the memory area, which will be accessed.

2.2.4 Proposal

We propose an XPP integration as an asynchronously pipelined functional unit for the RISC. We further propose an explicitly preloaded cache for the IRAMs, on top of the memory hierarchy existing within the RISC. Additionally a de-centralized explicitly preloaded configuration cache within the PAE array is employed to support preloading of configurations and fast switching between configurations.

Since the IRAM content is an explicitly preloaded memory area, a virtually unlimited number of such IRAMs can be used. They are identified by their memory address and their size. The IRAM content is explicitly preloaded by the application. Caching will increase performance by reusing data from the memory hierarchy. The cached operation also eliminates the need for explicit store instructions; they are handled implicitly by cache write back operations but can also be forced for synchronization.

The pipeline stages of the XPP functional unit are Load, Execute and Write Back (Store). The store is executed delayed as a cache write back. The pipeline stages execute in an asynchronous fashion, thus hiding the variable delays from the cache preloads and the PAE array.

The XPP functional unit is decoupled of the RISC by a FIFO, which is fed with the XPP instructions. At the head of this FIFO, the XPP PAE consumes and executes the configurations and the preloaded IRAMs. Synchronization of the XPP and the RISC is done explicitly by a synchronization instruction.

Instructions

In the following we define the instruction formats needed for the proposed architecture. We use a C style prototype definition to specify data types. All instructions, except the XPPSync instruction execute asynchronously. The XPPSync instruction can be used to force synchronization.

XPPPreloadConfig (void *ConfigurationStartAddress)

The configuration is added to the preload FIFO to be loaded into the configuration cache within the PAE array.

Note that speculative preloads are possible, since successive preload commands overwrite the previous.
The parameter is a pointer register of the RISC pointer register file. The size is implicitly contained in the configuration.

XPPPreload (int IRAM, void *StartAddress, int Size)
XPPPreloadClean (int IRAM, void *StartAddress, int Size)

This instruction specifies the contents of the IRAM for the next configuration. In fact, the memory area is added to the preload FIFO to be loaded into the specified IRAM.

The first parameter is the IRAM number. This is an immediate (constant) value.
The second parameter is a pointer to the starting address. This parameter is provided in a pointer register of the RISC pointer register file.
The third parameter is the size. This is an integer value. It resides in a general-purpose register of the RISC's integer register file.

The first variant actually preloads the data from memory.
The second variant is for write-only accesses. It skips the loading operation. Thus no cache misses can occur for this IRAM. Only the address and size are defined. They are obviously needed for the write back operation of the IRAM cache.

Note that speculative preloads are possible, since successive preload commands to the same IRAM overwrite each other. Thus only the last preload command is actually effective, when the configuration is executed.

XPPExecute ()

This instruction executes the last preloaded configuration with the last preloaded IRAM contents. Actually a configuration start command is issued to the FIFO. Then the FIFO is advanced; this means that further preload commands will specify the next configuration or parameters for the next configuration. Whenever a configuration finishes, the next one is consumed from the head of the FIFO, if its start command has already been issued.

XPPSync (void *StartAddress, int Size)

This instruction forces write back operations for all IRAMs that overlap the given memory area. If overlapping IRAMs are still in use by a configuration or preloaded to be used, this operation will block. Giving an address of NULL (zero) and a size of MAX_INT (bigger than the actual memory), this instruction can also be used to wait until all issued configurations finish.

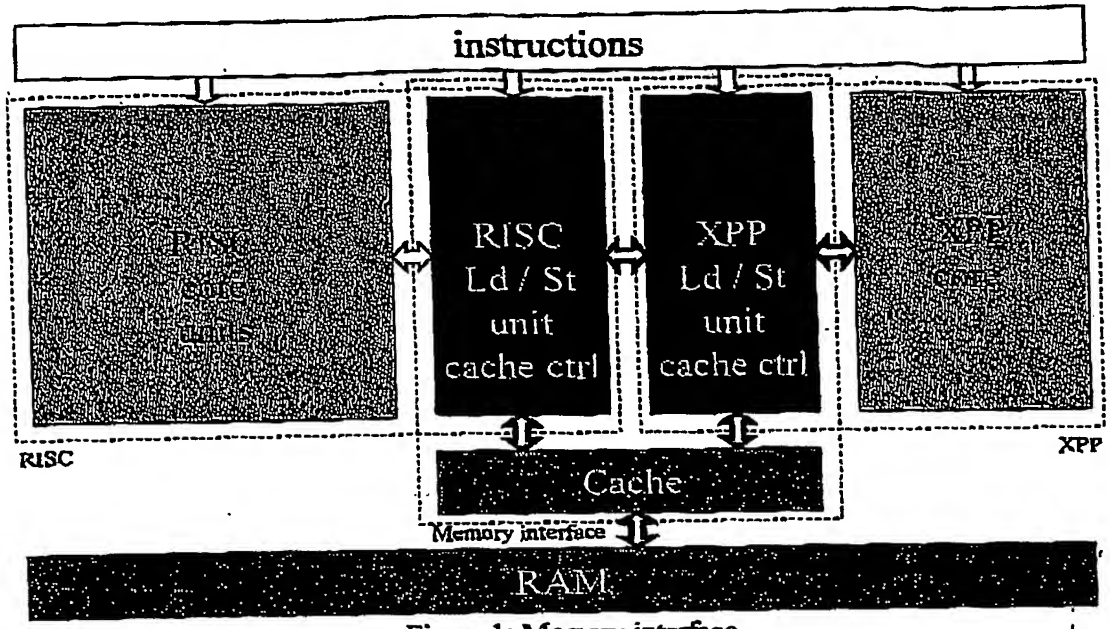
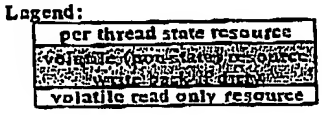


Figure 1: Memory interface

The XPP core shares the memory hierarchy with the RISC core using a special cache controller.

```
for (int i=0; i <1000; ++i) {
 XPPPreloadConfig( CONFIG1 );
 XPPPreload( IRAM2, 0x20000, 30 );
 XPPPreload( IRAM0, 0x20400, 200 );
 XPPPreloadClean( IRAM5, 0x20000, 10 );
/*
 other RISC computations
 in the meanwhile the burst preloads and
 the previous configuration are running
*/
 XPPExecute( CONFIG1 );
/*
 other RISC computations
 maybe burst preloads of
 another configuration and other data
*/
}
```

Note: in all places where constants are used, the value should actually come from a register



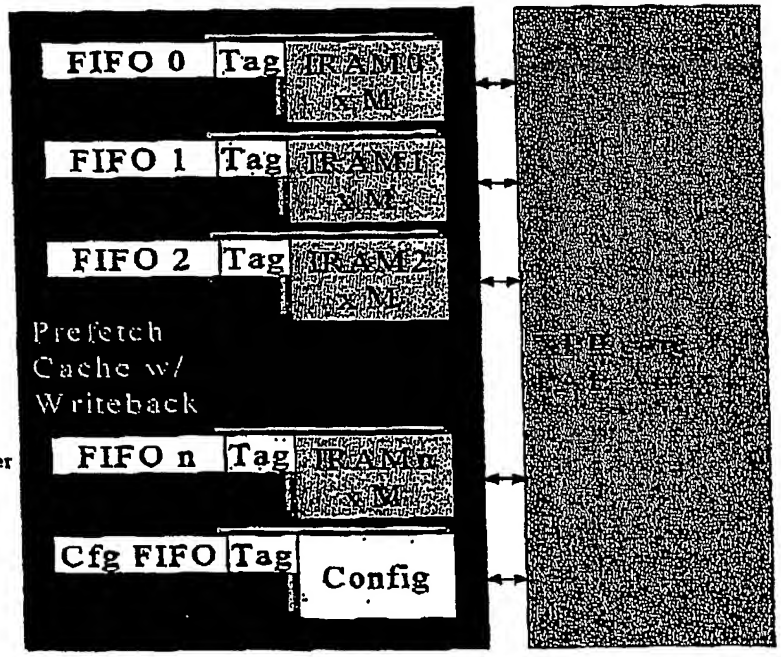


Figure 2 IRAM & configuration cache controller data structures and usage example

The preload-FIFOs in the above figure contain the addresses and sizes for already issued IRAM preloads, exposing them to the XPP cache controller. The FIFOs have to be duplicated for every virtual processor in an SMT environment. Tag is the typical tag for a cache line containing starting address, size and state (*empty / clean / dirty / in-use*). The additional *in-use* state signals usage by the current configuration. The cache controller cannot manipulate these IRAM instances.
The execute configuration command advances all preload FIFOs, copying the old state to the newly created entry. This way the following preloads replace the previously used IRAMs and configurations.

If no preload is issued for an IRAM, the previous information is retained, eliminating identical preload commands.

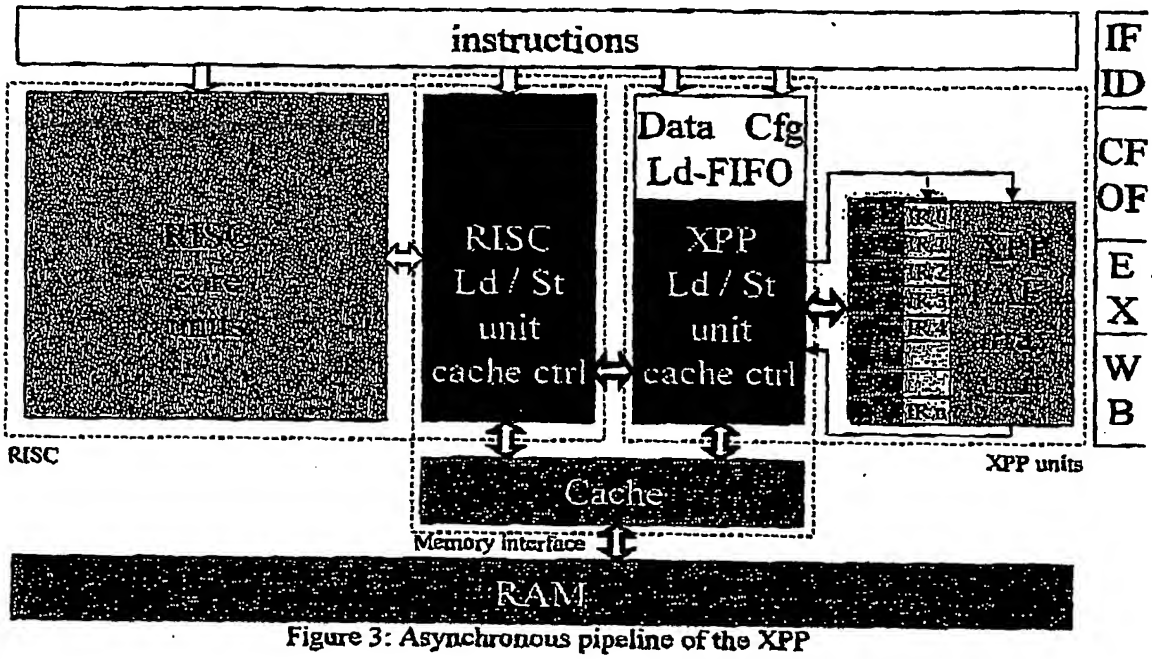


Figure 3: Asynchronous pipeline of the XPP

Each configuration's execute command has to be delayed (stalled) until all necessary preloads are finished, either explicitly by the use of a synchronization command or implicitly by the cache controller. Hence the cache controller (XPP Ld/St unit) has to handle the synchronization and execute commands as well, actually starting the configuration as soon as all data is ready. After the termination of the configuration, dirty IRAMs are written back to memory as soon as possible, if their content is not reused in the same IRAM. The XPP PAE array and the XPP cache controller can therefore be seen as a single unit since they do not have different instruction streams: rather, the cache controller can be seen as the configuration fetch (CF), operand fetch (OF) (IRAM preload) and write back (WB) stage of the XPP pipeline, also triggering the execute stage (EX) (PAE array).

Due to the long latencies, and their non-predictability (cache misses, variable length configurations), the stages can be overlapped several configurations wide using the configuration and data preload FIFO (=pipeline) for loose coupling. So if a configuration is executing and the data for the next has already been preloaded, the data for the next but one configuration is preloaded. These preloads can be speculative; the amount of speculation is the compiler's trade-off. The length of the preload FIFO can be several configurations; it is limited by diminishing returns, algorithm properties and the compiler's ability to schedule preloads early. Due to this loosely coupled operation, the interlocking cannot be done optimally by software (scheduling), but has to be enforced by hardware (hardware interlocking). Hence the XPP cache controller and the XPP PAE array can be seen as separate but not totally independent functional units.

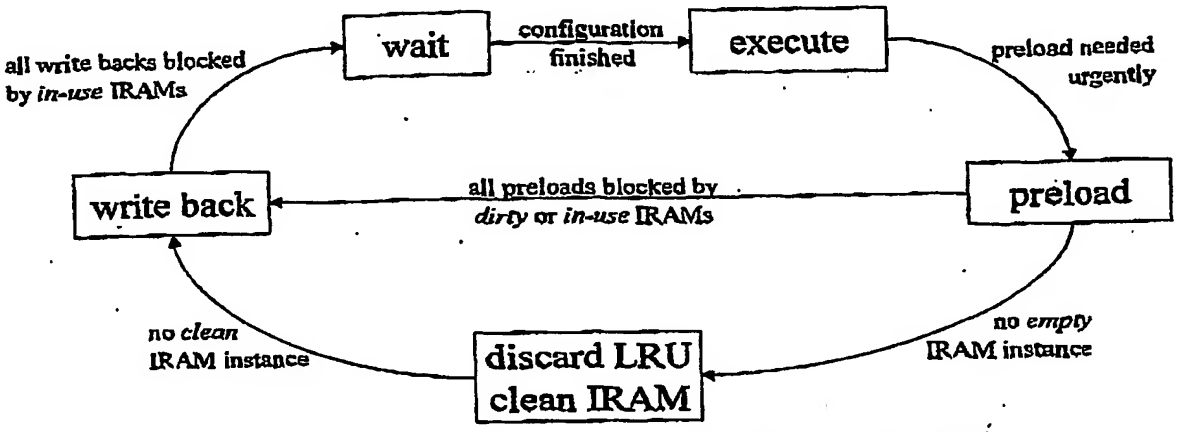


Figure 4: State transition diagram for the XPP cache controller

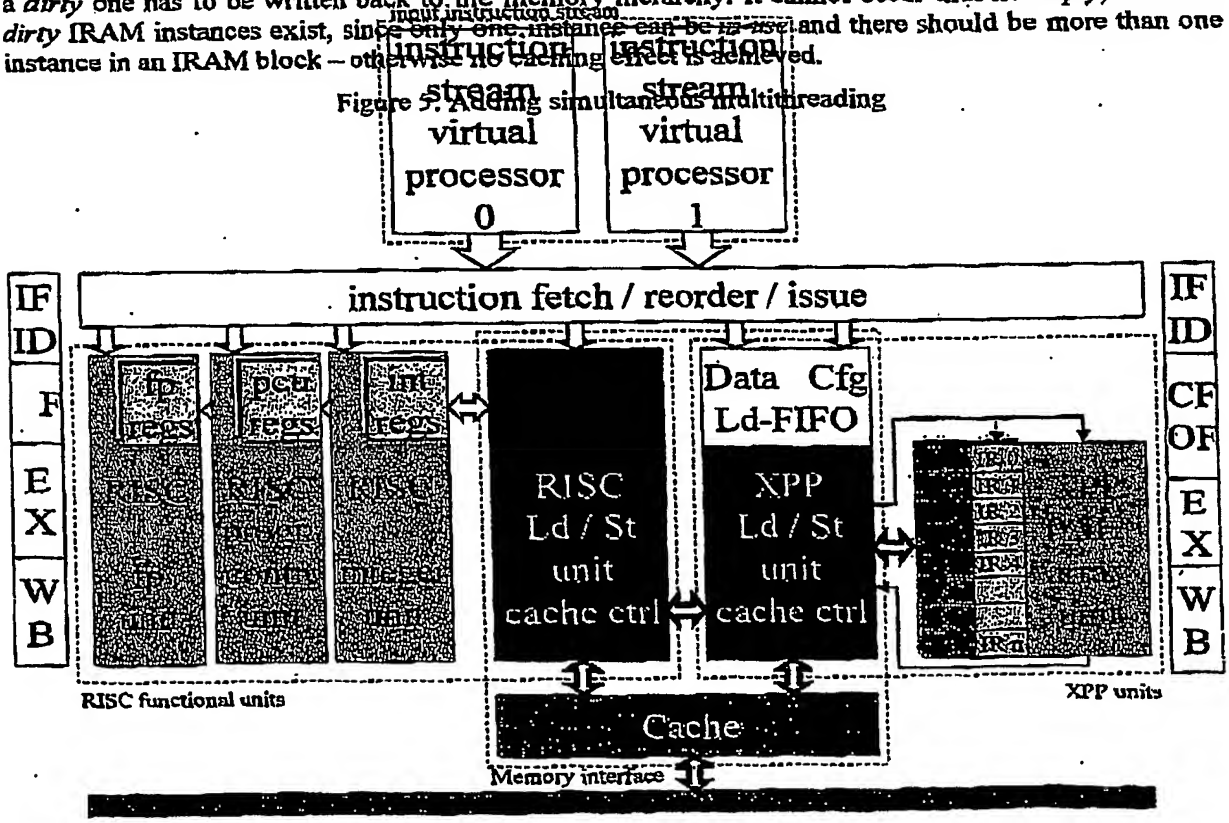
The XPP cache controller has several tasks. These are depicted as states in the above diagram. State transitions take place along the edges between states, whenever the condition for the edge is true. As soon as the condition is not true any more, the reverse state transition takes place. The activities for the states are as follows:

At the lowest priority, the XPP cache controller has to fulfill already issued preload commands, while writing back dirty IRAMs as soon as possible.

As soon as a configuration finishes, the next configuration can be started. This is a more urgent task than write backs or future preloads. To be able to do that, all associated yet unsatisfied preloads have to be finished first. Thus they are preloaded with the high priority inherited from the execute state.

A preload in turn can be blocked by an overlapping *in-use* or *dirty* IRAM instance in a different block or by the lack of *empty* IRAM instances in the target IRAM block. The former can be resolved by waiting for the configuration to finish and / or by a write back. To resolve the latter, the least recently used *clean* IRAM can be discarded, thus becoming *empty*. If no *empty* or *clean* IRAM instance exists, a *dirty* one has to be written back to the memory hierarchy. It cannot occur that no *empty*, *clean* or *dirty* IRAM instances exist, since only one instance can be *in-use* and there should be more than one instance in an IRAM block – otherwise no caching effect is achieved.

Figure 5: Adding simultaneous multithreading



In an SMT environment the load FIFOs have to be replicated for every virtual processor. The pipelines of the functional units are fed from the common fetch / reorder / issue stage. All functional units execute in parallel. Different units can execute instructions of different virtual processors.

2.2.5 Further Improvements

Write Pointer

To further decrease the penalty for unloaded IRAMs, a simple write pointer may be used per IRAM, which keeps track of the last address already in the IRAM. Thus no stall is required, unless an access beyond this write pointer is encountered. This is especially useful, if all IRAMs have to be reloaded after a task switch: The delay to the configuration start can be much shorter, especially, if the preload engine of the cache controller chooses the blocking IRAM next whenever several IRAMs need reloading

Longer FIFOs

The frequency at the bottom of the memory hierarchy (main memory) cannot be raised to the same extent as the frequency of the CPU core. To increase the concurrency between the RISC core and the PACT XPP core, the prefetch FIFOs in the above drawing can be extended. Thus the IRAM contents for several configurations can be preloaded, like the configurations themselves. A simple convention makes clear which IRAM preloads belong to which configuration: the configuration execute switches to the next configuration context. This can be accomplished by advancing the FIFO write pointer with every configuration execute, while leaving it unchanged after every preload. Unassigned IRAM FIFO entries keep their contents from the previous configuration, so every succeeding configuration will use the preceding configuration's IRAMx if no different IRAMx was preloaded.

If none of the memory areas to be copied to IRAMs is in any cache, extending the FIFOs does not help, as the memory *is* the bottleneck. So the cache size should be adjusted together with the FIFO length.

A drawback of extending the FIFO length is the increased likelihood that the IRAM content written by an earlier configuration is reused by a later one in another IRAM. A cache coherence protocol can clear the situation. Note however that the situation can be resolved more easily: If an overlap between any new IRAM area and a currently dirty IRAM contents of another IRAM bank is detected, the new IRAM is simply not loaded until the write back of the changed IRAM has finished. Thus the execution of the new configuration is delayed until the correct data is available.

For a short (single entry) FIFO, an overlap is extremely unlikely, since the compiler will usually leave the output IRAM contents of the previous configuration in place for the next configuration to skip the preload. The compiler does so using a coalescing algorithm for the IRAMs / vector registers.

Data Distribution

The IRAMs are block-oriented structures, which can be read in any order by the PAE array. However, the address generation adds complexity, reducing the number of PAEs available for the actual computation. So it is best, if the IRAMs are accessed in linear order. The memory hierarchy is block oriented as well, further encouraging linear access patterns in the code to avoid cache misses.

As the IRAM read ports limit the bandwidth between each IRAM and the PAE array to one word read per cycle, it can be beneficial to distribute the data over several IRAMs to remove this bottleneck. The top of the memory hierarchy is the source of the data, so the amount of cache misses never increases when the access pattern is changed, as long as the data locality is not destroyed.

Many algorithms access memory in linear order by definition to utilize block reading and simple address calculations. In most other cases and in the cases where loop tiling is needed to increase the data bandwidth between the IRAMs and the PAE array, the code can be transformed in a way that data is accessed in optimal order. In many of the remaining cases, the compiler can modify the access pattern by data layout rearrangements, so that finally the data is accessed in the desired pattern. If none of these optimizations can be used because of dependencies, or because the data layout is fixed, there are still two possibilities to improve performance:

Data Duplication:

Data is duplicated in several IRAMs. This circumvents the IRAM read port bottleneck, allowing several data items to be read from the input every cycle.

Several options are possible with a common drawback: data duplication can only be applied to input data: output IRAMs obviously cannot have overlapping address ranges.

- Using several IRAM preload commands specifying just different target IRAMs:

  This way cache misses occur only for the first preload. All other preloads will take place without cache misses – only the time to transfer the data from the top of the memory hierarchy to the IRAMs is needed for every additional load. This is only beneficial, if the cache misses plus the additional transfer times do not exceed the execution time for the configuration.

- Using an IRAM preload instruction to load multiple IRAMs concurrently:

  As identical data is needed in several IRAMs, they can be loaded concurrently by writing the same values to all of them. This amounts to finding a clean IRAM instance for every target IRAM, connecting them all to the bus and writing the data to the bus. The problem with this instruction is that it requires a bigger immediate field for the destination (16 bits instead of 4 for the XPP 64). Accordingly this instruction format grows at a higher rate, when the number of IRAMs is increased for bigger XPP arrays.

  The interface of this instruction looks like:

  XPPPreloadMultiple (int IRAMS, void *StartAddress, int Size)

  This instruction behaves as the XPPPreload / XPPPreloadClean instructions with the exception of the first parameter:
  The first parameter is IRAMS. This is an immediate (constant) value. The value is a bitmap – for every bit in the bitmap, the IRAM with that number is a target for the load operation.

  There is no "clean" version, since data duplication is applicable for read data only.

Data Reordering:

Data reordering changes the access pattern to the data only. It does not change the amount of memory that is read. Thus the number of cache misses stays the same.

- Adding additional functionality to the hardware:
  - Adding a vector stride to the preload instruction.

    A *stride* (displacement between two elements in memory) is used in vector load operations to load e.g.: a column of a matrix into a vector register.

    This is the most regular non-linear access pattern. It can be implemented in hardware by giving a stride to the preload instruction and adding the stride to the IRAM

identification state. One problem with this instruction is that the number of possible cache misses per IRAM load rises: In the worst case it can be one cache miss per loaded value, if the stride is equal to the cache line size and all data is not in the cache. But as already stated: the total number of misses stays the same – just the distribution changes. Still this is an undesirable effect.

The other problem is the complexity of the implementation and a possibly limited throughput, as the data paths between the layers of the memory hierarchy are optimized for block transfers. Transferring non-contiguous words will not use wide busses in an optimal fashion.

The interface of the instruction looks like:

XPPPreloadStride (int IRAM, void *StartAddress, int Size, int Stride)
XPPPreloadCleanStride (int IRAM, void *StartAddress, int Size, int Stride)

This instruction behaves as the XPPPreload / XPPPreloadClean instructions with the addition of another parameter:
The fourth parameter is the vector stride. This is an immediate (constant) value. It tells the cache controller, to load only every n^{th} value to the specified IRAM.

- Reordering the data at run time, introducing temporary copies.
  - On the RISC:

    The RISC can copy data at a maximum rate of one word per cycle for simple address computations and at a somewhat lower rate for more complex ones.

    With a memory hierarchy, the sources will be read from memory (or cache, if they were used recently) once and written to the temporary copy, which will then reside in the cache, too. This increases the pressure in the memory hierarchy by the amount of memory used for the temporaries. Since temporaries are allocated on the stack memory, which is re-used frequently, the chances are good that the dirty memory area is re-defined before it is written back to memory. Hence the write back operation to memory is of no concern.

  - Via an XPP configuration:

    The PAE array can read and write one value from every IRAM per cycle. Thus if half of the IRAMs are used as inputs and half of the IRAMs are used as outputs, up to eight (or more, depending on the number of IRAMs) values can be reordered per cycle, using the PAE array for address generation. As the inputs and outputs reside in IRAMs, it does not matter, if the reordering is done before or after the configuration that uses the data – the IRAMs can be reused immediately.

2.3 State of the XPP Core

As described in the previous section, the size of the state is crucial for the efficiency of context switches. However, although the size of the state is fixed for the XPP core, it depends on the declaration of the various state elements, whether they have to be saved or not.

The state of the XPP core can be classified as

1 Read only (instruction data)
   - configuration data, consisting of PAE configuration and routing configuration data

2 Read – Write
   - the contents of the data registers and latches of the PAEs, which are driven onto the busses
   - the contents of the IRAM elements

2.3.1 Limiting Memory Traffic

There are several possibilities to limit the amount of memory traffic during context switches.

Do not save read-only data

This avoids storing configuration data, since configuration data is read only. The current configuration is simply overwritten by the new one.

Save less data

If a configuration is defined to be uninterruptible (non pre-emptive), all of the local state on the busses and in the PAEs can be declared as scratch. This means that every configuration gets its input data from the IRAMs and writes its output data to the IRAMs. So after the configuration has finished all information in the PAEs and on the buses is redundant or invalid and does not have to be saved.

Save modified data only

To reduce the amount of R/W data, which has to be saved, we need to keep track of the modification state of the different entities. This incurs a silicon area penalty for the additional "dirty" bits.

Use caching to reduce the memory traffic

The configuration manager handles manual preloading of configurations. Preloading will help in parallelizing the memory transfers with other computations during the task switch. This cache can also reduce the memory traffic for frequent context switches, provided that a Least Recently Used (LRU) replacement strategy is implemented in addition to the preload mechanism.

The IRAMs can be defined to be local cache copies of main memory. Then each IRAM is associated with a starting address and modification state information. The IRAM memory cells should also be replicated as for the SMT support. Then only the starting addresses of the IRAMs have to be saved and restored as context. The starting addresses for the IRAMs of the current configuration select the IRAM instances with identical addresses to be used.

If no address tag of an IRAM instance matches the address of the newly loaded context, the corresponding memory area is loaded to an empty IRAM instance.

If no empty IRAM instance is available, a clean (unmodified) instance is declared empty (and hence must be reloaded later on).

If no clean IRAM instance is available, a modified (dirty) instance is cleaned by writing its data back to main memory. This adds a certain delay for the write back.

This delay can be avoided, if a separate state machine (cache controller) tries to clean inactive IRAM instances by using unused memory cycles to write back the IRAM instances' contents.

2.4 Context Switches

Usually a processor is viewed as executing a single stream of instructions. But today's multi tasking operating systems support hundreds of tasks being executed on a single processor. This is achieved by switching contexts, where all, or at least the most relevant parts of the processor state, which belong to the current task – the task's context – is exchanged with the state of another task, that will be executed next.

There are three types of context switches: switching of virtual processors with simultaneous multithreading (SMT, also known as HyperThreading), execution of an Interrupt Service Routine (ISR) and Task switch.

2.4.1 SMT Virtual Processor Switch

This type of context switch is executed without software interaction, totally in hardware. Instructions of several instruction streams are merged into a single instruction stream to increase instruction level parallelism and improve functional unit utilization. Hence the processor state cannot be stored to and reloaded from memory between instructions from different instruction streams: Imagine the worst case of alternating instructions from two streams and the hundreds to thousand of cycles needed to write the processor state to memory and read in another state.

Hence hardware designers have to replicate the internal state for every virtual processor. Every instruction is executed within the context (on the state) of the virtual processor, whose program counter was used to fetch the instruction. By replicating the state, only the multiplexers, which have to be inserted to select one of the different states, have to be switched.

Thus the size of the state also increases the silicon area needed to implement SMT, so the size of the state is crucial for many design decisions.

For the design presented in the previous sections, the state is minimal, thus enabling efficient implementation of simultaneous multithreading.

2.4.2 Interrupt Service Routine

This type of context switch is handled partially by hardware and partially by software. All of the state modified by the ISR has to be saved on entry and must be restored on exit.

The part of the state, which is destroyed by the jump to the ISR, is saved by hardware (e.g. PC). It is the ISR's responsibility to save and restore the state of all other resources, that are actually used within the ISR.

The more state information to be saved, the slower the interrupt response time will be and the greater the performance impact will be if external events trigger interrupts at a high rate.

The execution model of the instructions will also affect the tradeoff between short interrupt latencies and maximum throughput: Throughput is maximized if the instructions in the pipeline are finished, and the instructions of the ISR are chained. This adversely affects the interrupt latency. If, however, the instructions are abandoned (pre-empted) in favor of a short interrupt latency, they must be fetched again later, which affects throughput. The third possibility would be to save the internal state of the instructions within the pipeline, but this requires too much hardware effort, so this is not done normally.

2.4.3 Task Switch

This type of context switch is executed totally in software. All of a task's context (state) has to be saved to memory, and the context of the new task has to be reloaded. Since tasks are usually allowed to use all of the processor's resources to achieve top performance, all of the processor state has to be saved and restored. If the amount of state is excessive, the rate of context switches must be decreased by less frequent rescheduling, or a severe throughput degradation will result, as most of the time will be spent in saving and restoring task contexts. This in turn increases the response time for the tasks.

2.5 Software / Hardware Interface

According to the design parameter changes and the corresponding changes to the hardware, the hardware / software interface has changed. In the following the most prominent changes and their handling will be discussed:

2.5.1 Explicit Cache

The proposed cache is not a usual cache, which would be – not considering performance issues – invisible to the programmer / compiler, as its operation is transparent. The proposed cache is an explicit cache. Its state has to be maintained by software.

Cache Consistency & Pipelining of Preload / Configuration / Write back

The software is responsible for cache consistency. It is possible to have several IRAMs caching the same, or overlapping memory areas. As long as only one of the IRAMs is written, this is perfectly ok: Only this IRAM will be dirty and will be written back to memory. If however more than one of the IRAMs is written, it is not defined, which data will be written to memory. This is a software bug (non deterministic behavior).

As the execution of the configuration is overlapped with the preloads and write backs of the IRAMs, it is possible to create preload / configuration sequences, that contain data hazards. As the cache controller and the XPP array can be seen as separate functional units, which are effectively pipelined, these data hazards are equivalent to pipeline hazards of a normal instruction pipeline. As with any ordinary pipeline, there are two possibilities to resolve this:

- Hardware interlocking:

  Interlocking is done by the cache controller: If the cache controller detects, that the tag of a dirty or in-use item in IRAMx overlaps a memory area used for another IRAM preload, it has to stall that preload, effectively serializing the execution of the current configuration and the preload.

- Software interlocking:

  If the cache controller does not enforce interlocking, the code generator has to insert explicit synchronize instructions to take care of potential interlocks. Inter- procedural and inter-modular alias- and data- dependency analyses can determine if this is the case, while scheduling algorithms help to alleviate the impact of the necessary synchronization instructions.

In either case, as well as in the case of pipeline stalls due to cache misses, SMT can use the computation power, that would be wasted otherwise.

Code Generation for the Explicit Cache:

Apart from the explicit synchronization instructions issued with software interlocking, the following instructions have to be issued by the compiler.

- Configuration preload instructions, preceding the IRAM preload instructions, that will be used by that configuration. These should be scheduled as early as possible by the instruction scheduler.
- IRAM preload instructions, which, too, should be scheduled as early as possible by the instruction scheduler.
- Configuration execute instructions, following the IRAM preload instructions for that configuration. These instructions should be scheduled between the estimated minimum and the estimated maximum of the cumulative latency of their preload instructions.

Asynchronicity to Other Functional Units

A configuration wait instruction followed by an instruction forcing a cache write back must be issued by the compiler, if an instruction of another functional unit (mainly the Ld/St unit) can access a memory area, that is potentially dirty or in-use in an IRAM. This forces a synchronization of the instruction streams and the cache contents, avoiding data hazards. A thorough inter-procedural and inter-modular array alias analysis limits the frequency of these synchronization instructions to an acceptable level.

3 Program Optimizations

3.1 Code Analysis

In this section we describe the analyses that can be performed on programs. These analyses are then used by different optimizations. They describe the relationships between data and memory locations in the program. More details can be found in several books [2,3,5].

3.1.1 Data-Flow Analysis

Data-flow analysis examines the flow of scalar values through a program, to provide information about how the program manipulates its data. This information can be represented by dataflow equations that have the following general form for object *i*, that can be an instruction or a basic block, depending on the problem to solve:

$$Ex[i] = Prod[i] \cup (In[i] - Supp[i])$$

It means that data available at the end of the execution of object *i*, *Ex[i]*, are either produced by *i*, *Prod[i]* or were alive at the beginning of *i*, *In[i]*, but were not deleted during the execution of *i*, *Supp[i]*.

These equations can be used to solve several problems like:

- the problem of reaching definitions,
- the Def-Use and Use-Def chains, describing respectively for a definition all uses that can be reached from it, and for a use all definitions that can reach it.
- the available expressions at a point in the program,
- the live variables at a point in the program,

whose solutions are then used by several compilation phases, analysis, or optimizations.

As an example let us take the problem of computing the Def-Use chains of the variables of a program. This information can be used for instance by the data dependence analysis for scalar variables or by the register allocation. A Def-Use chain is associated to each definition of a variable and is the set of all visible uses from this definition. The data-flow equations presented above are applied to the basic blocks to detect the variables that are passed from one block to another along the control-flow graph. In the figure below, two definitions for variable x are produced: *S1* in *B1* and *S4* in *B3*. Hence the variable that can be found at the exit of *B1* is *Ex(B1)={x(S1)}*, and at the exit of *B4* is *Ex(B4)={x(S4)}*. Moreover we have *Ex(B2)=Ex(B1)* as no variable is defined in *B2*. Using these sets, we find that the uses of x in *S2* and *S3* depend on the definition of x in *B1*, that the use of x in *S5* depend on the definitions of x in *B1* and *B3*. The Def-use chains associated with the definitions are then $D(S1) = \{S2, S3, S5\}$ and $D(S4) = \{S5\}$.

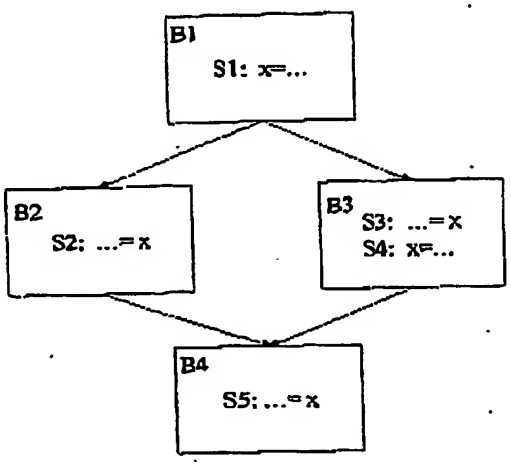


Figure 6:Control-flow graph of a piece of program

3.1.2 Data Dependence Analysis

A data dependence graph represents the dependences existing between operations writing or reading the same data. This graph is used for optimizations like scheduling, or certain loop optimizations to test their semantic validity. The nodes of the graph represent the instructions, and the edges represent the data dependences. These dependences can be of three types: true (or flow) dependence when a variable is written before being read, anti-dependence when a variable is read before being written, and output dependence when a variable is written twice. Here is a more formal definition [3].

Definition

Let S and S' be 2 statements, then S' depends on S, noted $S\ \delta\ S'$ iff:

(1) S is executed before S'

(2) $\exists v \in VAR : v \in DEF(S) \mathrm{I}\ USE(S') \vee v \in USE(S) \mathrm{I}\ DEF(S') \vee v \in DEF(S) \mathrm{I}\ DEF(S')$

(3) There is no statement T such that S is executed before T and T is executed before S', and $v \in DEF(T)$

Where *VAR* is the set of the variables of the program, *DEF(S)* is the set of the variables defined by instruction *S*, and *USE(S)* is the set of variables used by instruction *S*.

Moreover if the statements are in a loop, a dependence can be loop-independent or loop-carried. This notion introduces the definition of the distance of a dependence. When a dependence is loop-independent it means that it occurs between two instances of different statements in the same iteration, and then its distance is equal to 0. On the contrary when a dependence occurs between two instances in two different iterations the dependence is loop-carried, and the distance is equal to the difference between the iteration numbers of the two instances.

The notion of direction of dependence generalizes the notion of distance, and is generally used when the distance of a dependence is not constant, or cannot be computed with precision. The direction of a dependence is given by $<$, if the dependence between S and S' occurs when the instance of S is in an iteration before the iteration of the instance of S', $=$ if the two instances are in the same iteration, and $>$ if the instance of S is an iteration after the iteration of the instance of S'.

In the case of a loop nest, we have then distance and direction vector, with one element for each level of the loop nest. The figures below illustrate all these definitions. The data dependence graph is used by a lot of optimizations, and is also useful to determine if their application is valid. For instance a loop can be vectorized if its data dependence graph does not contain any cycle.

```
for(i=0; i<N; i=i+1) {
S:   a[i] = b[i] + 1;
S1:  c[i] = a[i] + 2;
}
```

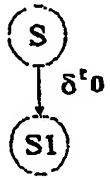


Figure 7: Example of a true dependence with distance 0 on array a

```
for(i=0; i<N; i=i+1) {
 S:  a[i] = b[i] + 1;
 S1  b[i] = c[i] + 2;
}
```

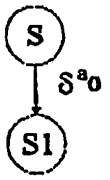


Figure 8: Example of an anti-dependence with distance 0 on array b

```
for(i=0; i<N; i=i+1) {
S:    a[i] = b[i] + 1;
S1:   a[i] = c[i] + 2;
}
```

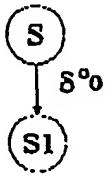


Figure 9: Example of an output dependence with distance 0 on array a

```
for(j=0;j<=N;j++)
  for(i=0;i<=N;i++)
     {
S1:   c[i][j] = 0;
      for(k=0;k<=N;k++)
S2:      c[i][j] = c[i][j] + a[i][k]*b[k][j];
      }
```

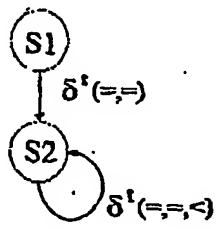


Figure 10: Example of a dependence with direction vector(=,=) between S1 and S2 and a dependence with direction vector (=,=,<) between S2 and S2.

```
for(i=0;i<=N;i++)
   for(j=0;j<=N;j++)
S:    a[i][j] = a[i][j+2] + b[i];
```

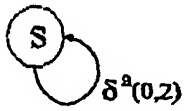


Figure 11: Example of an anti-dependence with distance vector (0,2).

3.1.3 Interprocedural Alias Analysis

The aim of alias analysis is to determine if a memory location is aliased by several objects, like variables or arrays, in a program. It has a strong impact on data dependence analysis and on the application of code optimizations. Aliases can occur with statically allocated data, like unions in C where all fields refer to the same memory area, or with dynamically allocated data, which are the usual targets of the analysis. In Figure 12, we have a typical case of aliasing where *p alias b*.

```
int b[100],*p;

for(p=b;p < &b[100];p++)
   *p=0;
```

Figure 12: Example for typical aliasing

Alias analysis can be more or less precise depending on whether or not it takes the control-flow into account. When it does, it is called flow-sensitive, and when it does not, it is called flow-insensitive. Flow-sensitive alias analysis is able to detect in which blocks along a path two objects are aliased. As it is more precise, it is more complicated and more expensive to compute. Usually flow-insensitive alias information is sufficient. This aspect is illustrated in Figure 13 where a flow-insensitive analysis would find that *p alias b*, but where a flow-sensitive analysis would be able to find that *p alias b* only in block *B2*.

Furthermore aliases are classified into must-aliases and may-aliases. For instance, if we consider flow-insensitive may-alias information, then *x alias y*, iff *x* and *y* may, possibly at different times, refer to the same memory location. And if we consider flow-insensitive must-alias information, *x alias y*, iff *x*

and *y* must, throughout the execution of a procedure, refer to the same storage location. In the case of Figure 13, if we consider flow-insensitive may-alias information, *p alias b* holds, whereas if we consider flow-insensitive must-alias information, *p alias b* does not hold. The kind of information to use depends on the problem to solve. For instance, if we want to remove redundant expressions or statements, must-aliases must be used, whereas if we want to build a data dependence graph may-aliases are necessary.

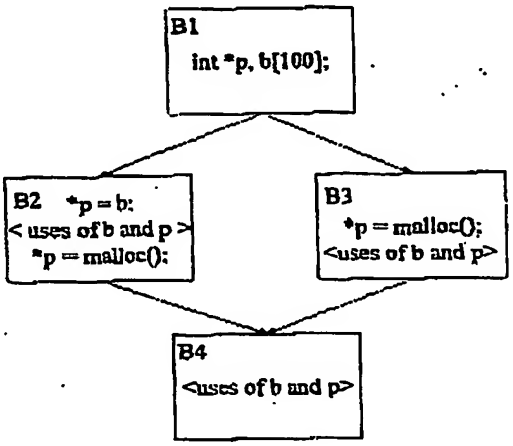


Figure 13:Example of control-flow sensitivity

Finally this analysis must be interprocedural to be able to detect aliases caused by non-local variables and parameter passing. The latter case is depicted in Figure 14 where *i* and *j* are aliased through the function call where *k* is passed twice as parameter.

```
void foo(int *i,int* j)
{
  *i = *j+1;
}
...
foo(&k,&k);
```

Figure 14: Example for aliasing by parameter passing

3.1.4 Interprocedural Value Range Analysis

This analysis can find the range of values taken by the variables. It can help to apply optimizations like dead code elimination, loop unrolling and others. For this purpose it can use information on the types of variables and then consider operations applied on these variables during the execution of the program. Thus it can determine for instance if tests in conditional instruction are likely to be met or not, or determine the iteration range of loop nests.

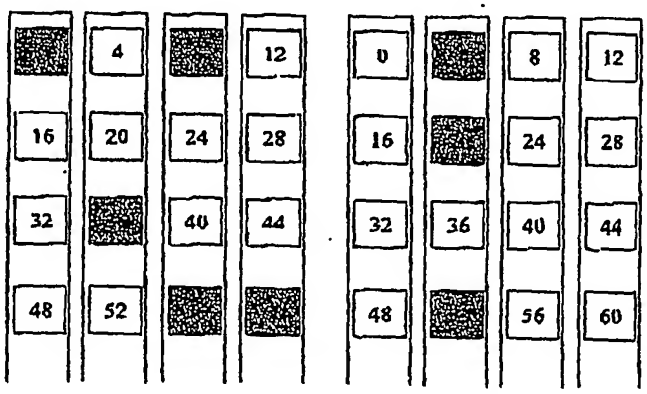
This analysis has to be interprocedural as for instance loop bounds can be passed as parameters of a function, like in the following example. We know by analyzing the code that in the loop executed with array *a*, *N* is at least equal to 11, and that in the loop executed with array *b*, *N* is at most equal to 10.

```
void foo(int *c,int N)
{
  int i;

 for (i=0;i<N;i++)
    c[i] = g(i,2);
}

...

if (N > 10)
   foo(a,N);
else
   foo(b,N);
```

The value range analysis can be supported by the programmer by giving further value constraints which cannot be retrieved from the language semantics. This can be done by pragmas or a compiler known assert function.

3.1.5 Alignment Analysis

Alignment analysis deals with data layout for distributed memory architectures. As stated by Saman Amarasinghe: "Although data memory is logically a linear array of cells, its realization in hardware can be viewed as a multi-dimensional array. Given a dimension in this array, alignment analysis will identify memory locations that always resolve to a single value in that dimension. For example, if the dimension of interest is memory banks, alignment analysis will identify if a memory reference always accesses the same bank". This is the case in the second part of the figure below, that can be found in [10], where all accesses, depicted in blue, occur to the same memory bank, whereas in the first part, the accesses are not aligned. He adds then that: "Alignment information is useful in a variety of compiler-controlled memory optimizations leading to improvements in programmability, performance, and energy consumption."



Alignment analysis, for instance, is able to help find a good distribution scheme of the data and is furthermore useful for automatic data distribution tools. An automatic alignment analysis tool can be able to automatically generate alignment proposals for the arrays accessed in a procedure and thus simplifies the data distribution problem. This can be extended with an interprocedural analysis taking into account dynamic realignment.

Alignment analysis can also be used to apply loop alignment that transforms the code directly rather than the data layout in itself, as shown later. Another solution can be used for the PACT XPP, relying on the fact that it can handle aligned code very efficiently. It consists in adding a conditional instruction testing if the accesses in the loop body are aligned followed by the necessary number of peeled iterations of the loop body, then the aligned loop body, and then some compensation code. Only the aligned code is then executed by the PACT XPP, the rest is executed by the host processor. If the alignment analysis is more precise (inter-procedural or inter-modular) less conditional code has to be inserted.

3.2 Code Optimizations

Most of the optimizations and transformations presented here can be found in detail in [4], and also in [2,3,5].

3.2.1 General Transformations

We present in this section a few general optimizations that can be applied to straightforward code, and to loop bodies. These are not the only ones that appear in a compiler, but they are mentioned in the sequel of this document.

Constant Propagation

This optimization propagates the values of constants into the expressions using them throughout the program. This way a lot of computations can be done statically by the compiler, leaving less work to be done during the execution, this part of the optimization is also known as constant folding.

```
N = 256;
c = 3;
for(i=0;i <= N;i++)
    a[i] = b[i] + c;
```

```
for(i=0; i<= 256; i++)
    a[i] = b[i] + 3;
```

Figure 15: Example of constant propagation

Copy Propagation

This optimization simplifies the code by removing redundant copies of the same variable in the code. These copies can be produced by the programmer himself or by other optimizations. This optimization reduces the register pressure and the number of register-to-register move instructions.

```
t = i*4;
r = t;
for(i=0;i <= N;i++)
    a[r] = b[r] + a[i];
```

```
t = i*4;
for(i=0;i <= N;i++)
    a[t] = b[t] + a[i];
```

Figure 16: Example of copy propagation

Dead Code Elimination

This optimization removes pieces of code that will never be executed. Code is never executed if it is in the branch of conditional statement whose condition is always evaluated to true or false, or if it is a loop body, whose number of iterations is always equal to 0.

Code updating variables, that are never used, is also useless and can be removed as well. If a variable is never used, then the code updating it and its declaration can also be eliminated.

```
for(i=0;i <= N;i++) {
    for(j=0;j<0;j++)
       a[j] = b[j] + a[i];
    for(j=0;j<10;j++)
        a[j+1] = a[j] + b[j];
}
```

```
for(i=0;i <= N;i++) {
    for(j=0;j<10;j++)
        a[j+1] = a[j] + b[j];
}
```

Figure 17: Example of dead code elimination

Forward Substitution

This optimization is a generalization of copy propagation. The use of a variable is replaced by its defining expression. It can be used for simplifying the data dependency analysis and the application of other transformations by making the use of loop variables visible.

```
c = N + 1;
for(i=0;i <= N;i++)
    a[c] = b[c] + a[i];
```

```
for(i=0; i<= N; i++)
    a[N+1] = b[N+1] + a[i];
```

Figure 18: Example of forward substitution

Idiom Recognition

This transformation recognizes pieces of code and can replace them by calls to compiler known functions, or less expensive code sequences, like code for absolute value computation.

```
for(i=0; i<N; i++){
    c = a[i] - b[i];
    if (c<0)
       c = -c;
    d[i] = c;
}
```

```
for(i=0; i<N; i++){
    c = a[i] - b[i];
    c = abs(c);
    d[i] = c;
}
```

Figure 19: Example of idiom recognition.

3.2.2 Loop Transformations

Loop Normalization

This transformation ensures that the iteration space of the loop is always with a lower bound equal to 0 or 1 (depending on the input language), and with a step of 1. The array subscript expressions and the bounds of the loops are modified accordingly. It can be used before loop fusion to find opportunities, and ease inter-loop dependence analysis, and it also enables the use of dependence tests that needs normalized loop to be applied.

```
for(i=2; i<N; i=i+2)
    a[i] = b[i];
```

```
for(i=0; i<(N-2)/2; i++)
    a[2*i+2] = b[2*i+2];
```

Figure 20: Example of loop normalization

Loop Reversal

This transformation changes the direction in which the iteration space of a loop is scanned. It is usually used in conjunction with loop normalization and other transformations, like loop interchange, because it changes the dependence vectors.

```
for(i=N; i>=0; i--)
    a[i] = b[i];
```

```
for(i=0; i<=N; i++)
    a[i] = b[i];
```

Figure 21: Example of loop reversal

Strength Reduction

This transformation replaces expressions in the loop body by equivalent but less expensive ones. It can be used on induction variables, other than the loop variable, to be able to eliminate them.

```
for(i=0; i<N; i++)
    a[i] = b[i] + c*i;
```

```
t = c;
for(i=0; i<N; i++){
    a[i] = b[i] + t;
    t = t + c;
}
```

Figure 22: Example of strength reduction

Induction Variable Elimination

This transformation can use strength reduction to remove induction variables from a loop, hence reducing the number of computations and easing the analysis of the loop. This also removes dependence cycles due to the update of the variable, enabling vectorization.

```
for(i=0; i<=N; i++) {
    k = k + 3;
    a[i] = b[i] + a[k];
}
```

```
for(i=0; i<=N; i++){
    a[i] = b[i] + a[k+(i+1)*3];
}

k = k +(N+1)*3;
```

Figure 23: Example of induction variable elimination

Loop-Invariant Code Motion

This transformation moves computations outside a loop if their result is the same in all iterations. This allows to reduce the number of computations in the loop body. This optimization can also be conducted in the reverse fashion in order to get perfectly nested loops, that are easier to handle by other optimizations.

```
for(i=0; i<N; i++)
    a[i] = b[i] + x*y;
```

```
if (N >= 0)
    c = x*y;
for(i=0; i<N; i++)
    a[i] = b[i] + c;
```

Figure 24: Example of loop-invariant code motion

Loop Unswitching

This transformation moves a conditional instruction outside of a loop body if its condition is loop-invariant. The branches of the condition are then made of the original loop with the appropriate original statements of the conditional statement. It allows further parallelization of the loop by removing control-flow in the loop body and also removing unnecessary computations from it.

```
for(i=0; i<N; i++) {
    a[i] = b[i] + 3;
    if (x > 2)
        b[i] = c [i] + 2;
    else
        b[i] = c[i] - 2;
}
```

```
if (x > 2)
   for(i=0; i<N; i++){
      a[i] = b[i] + 3;
      b[i] = c[i] + 2;
   }
else
   for(i=0; i<N; i++){
      a[i] = b[i] + 3;
      b[i] = c[i] - 2;
   }
```

Figure 25: Example of loop unswitching

If-Conversion

This transformation is applied on loop bodies with conditional instructions. It changes control dependences into data dependences and allows then vectorization to take place. It can be used in conjunction with loop unswitching to handle loop bodies with several basic blocks. The conditions, where array expressions could appear, are replaced by boolean terms called guards. Processors with predicated execution support can execute directly such code.

```
for(i = 0;i < N; i++) {
    a[i] = a[i] + b[i];
    if (a[i] != 0)
       if (a[i] > c[i])
          a[i] = a[i] - 2;
       else
          a[i] = a[i] + 1;
    d[i] = a[i] * 2;
}
```

```
for(i = 0;i < N;i++) {
    a[i] = a[i] + b[i];
    c2 = (a[i] != 0);
    if (c2) c4 = (a[i] > c[i]);
    if (c2 && c4) a[i] = a[i] - 2;
    if (c2 && !c4) a[i] = a[i] + 1;
    d[i] = a[i] * 2;
}
```

Figure 26: Example of if-conversion

Strip-Mining

This transformation enables to adjust the granularity of an operation. It is commonly used to choose the number of independent computations in the inner loop nest. When the iteration count is not known at compile time, it can be used to generate a fixed iteration count inner loop satisfying the resource constraints. It can be used in conjunction with other transformations like loop distribution or loop interchange. It is also called loop sectioning. Cycle shrinking, also called stripping, is a specialization of strip-mining.

```
for(i=0; i<N; i++)
    a[i] = b[i] + c;
```

```
up = (N/16)*16;
for(i=0; i<up; i = i + 16)
    a[i:i+16] = b[i:i+16] + c;
for(j=i+1;j<N;j++)
    a[i] = b[i] + c;
```

Figure 27: Example of strip-mining

Loop Tiling

This transformation modifies the iteration space of a loop nest by introducing loop levels to divide the iteration space in tiles. It is a multi-dimensional generalization of strip-mining. It is generally used to improve memory reuse, but can also improve processor, register, TLB, or page locality. It is also called loop blocking.

The size of the tiles of the iteration space is chosen so that the data needed in each tile fit in the cache memory, thus reducing the cache misses. In the case of coarse-grain computers, the size of the tiles can also be chosen so that the number of parallel operations of the loop body fit the number of processors of the computer.

```
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        a[i][j] = b[j][i];
```

```
for(ii=0; ii<N; ii = ii+16)
   for(jj=0; jj<N; jj = jj+16)
      for(i=ii; i< min(ii+15,N); i++)
         for(j=jj; j< min(jj+15,N); j++)
            a[i][j] = b[j][i];
```

Figure 28: Example of loop tiling

Loop Interchange

This transformation is applied to a loop nest to move inside or outside (depending on the searched effect) the loop level containing data dependences. It can:

- enable vectorization by moving inside an independent loop and outside a dependent loop, or
- improve vectorization by moving inside the independent loop with the largest range, or
- deduce the stride, or
- increase the number of loop-invariant expressions in the inner-loop, or
- improve parallel performance by moving an independent loop outside of a loop nest to increase the granularity of each iteration and reduce the number of barrier synchronizations.

```
for(i=0; i<N; i++)
    for(j=0;j<N; j++)
        a[i] = a[i] + b[i][j];
```

```
for(j=0; j<N; j++)
   for(i=0; i<N; i++)
      a[i] = a[i] + b[i][j];
```

Figure 29: Example of loop interchange

Loop Coalescing / Collapsing

This transformation combines a loop nest into a single loop. It can improve the scheduling of the loop, and also reduces the loop overhead. Collapsing is a simpler version of coalescing in which the number of dimensions of arrays is reduced as well. Collapsing reduces the overhead of nested loops and multi-dimensional arrays. Collapsing can be applied to loop nests that iterate over memory with a constant stride, otherwise loop coalescing is a better approach. It can be used to make vectorizing profitable by increasing the iteration range of the innermost loop.

```
for(i=0; i<N; i++)
    for(j=0;j<M; j++)
      a[i][j] = a[i][j] + c;
```

```
for(k=0; k<N*M; k++){
    i = ((k-1)/m)*m + 1;
    j = ((T-1)%m) + 1;
    a[i][j] = a[i][j] + c;
}
```

Figure 30: Example of loop coalescing

Loop Fusion

This transformation, also called loop jamming, merges 2 successive loops. It reduces loop overhead, increases instruction-level parallelism, improves register, cache, TLB or page locality, and improves the load balance of parallel loops. Alignment can be taken into account by introducing conditional instructions to take care of dependences.

```
for(i=0; i<N; i++)
    a[i] = b[i] + c;

for(i=0; i<N; i++)
    d[i] = e[i] + c;
```

```
for(i=0; i<N; i++) {
    a[i] = b[i] + c;
    d[i] = e[i] + c;
}
```

Figure 31: Example of loop fusion

Loop Distribution

This transformation, also called loop fission, allows to split a loop in several pieces in case the loop body is too big, or because of dependences. The iteration space of the new loops is the same as the iteration space of the original loop. Loop spreading is a more sophisticated distribution.

```
for(i=0; i<N; i++) {
    a[i] = b[i] + c;
    d[i] = e[i] + c;
}
```

```
for(i=0; i<N; i++)
    a[i] = b[i] + c;

for(i=0;i<N; i++)
    d[i] = e[i] +c;
```

Figure 32: Example of loop distribution

Loop Unrolling / Unroll-and-Jam

This transformation replicates the original loop body in order to get a larger one. A loop can be unrolled partially or completely. It is used to get more opportunity for parallelization by making the loop body bigger, it also improves register, or cache usage and reduces loop overhead. Loop unrolling the outer loop followed by merging the induced inner loops is referred to as unroll-and-jam.

```
for(i=0; i<N; i++)
    a[i] = b[i] + c
```

```
for(i=0; i<N; i = i+2){
    a[i] = b[i] + c;
    a[i+1] = b[i+1] + c;
}
if ((N-1)%2) == 1)
    a[N-1] = b[N-1] + c;
```

Figure 33: Example of loop unrolling

Loop Alignment

This optimization transforms the code to get aligned array accesses in the loop body. Its effect it to transform loop-carried dependences into loop-independent dependences, which allows to extract more parallelism from a loop. It can use different transformations, like loop peeling or introduce conditional statements, to achieve its goal. This transformation can be used in conjunction with loop fusion to enable this optimization by aligning the array accesses in both loop nests. In the example below, all accesses to array *a* become aligned.

```
for(i=2;i <= N;i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i-1] * 2;
    e[i] = a[i-1] + d[i+1];
}
```

```
for(i=1; i<=N; i++) {
    if (i>1) a[i] = b[i] + c[i];
    if (i<N) d[i+1] = a[i] * 2;
    if (i<N) e[i+1] = a[i] + d[i+2];
}
```

Figure 34: Example of loop alignment

Loop Skewing

This transformation is used to enable parallelization of a loop nest. It is useful in combination with loop interchange. It is performed by adding the outer loop index multiplied by a skew factor, *f*, to the bounds of the inner loop variable, and then subtracting the same quantity from every use of the inner loop variable inside the loop.

```
for(i=1; i <= N; i++)
    for(j=1;j <= N; j++)
        a[i] = a[i+j] + c;
```

```
for(i=1; i <= N; i++)
    for(j=i+1;j <= i+N; j++)
        a[i] = a[j] + c;
```

Figure 35: Example of loop skewing

Loop Peeling

This transformation removes a small number of beginning or ending iterations of a loop to avoid dependences in the loop body. These removed iterations are executed separately. It can be used for matching the iteration control of adjacent loops to enable loop fusion.

```
for(i=0; i<=N; i++)
    a[i][N] = a[0][N] + a[N][N];
```

```
a[0][N] = a[0][N] + a[N][N];
for(i=1;i<=N-1; i++)
    a[i][N] = a[0][N] + a[N][N];
a[N][N] = a[0][N] + a[N][N];
```

Figure 36: Example of loop peeling

Loop Splitting

This transformation cuts the iteration space in pieces by creating other loop nests. It is also called Index Set Splitting, and is generally used because of dependences that prevent parallelization. The iteration space of the new loops is a subset of the original one. It can be seen as a generalization of loop peeling.

```
for(i=0; i<=N; i++)
    a[i] = a[N-i+1] + c;
```

```
for(i=0;i<(N+1)/2; i++)
    a[i] = a[N-i+1] + c;
for(i= (N+1)/2;i <= N;i++)
    a[i] = a[N-i+1] + c;
```

Figure 37: Example of loop splitting

Node Splitting

This transformation splits a statement in pieces. It is used to break dependence cycles in the dependence graph due to the too high granularity of the nodes, thus enabling vectorization of the statements.

```
for(i=0;i < N;i++) {
    b[i] = a[i] + c[i] * d[i] ;
    a[i+1] = b[i] *. (d[i] - c[i]);
}
```

```
for(i = 0,i < N;i++) {
    t1[i] = c[i] * d[i];
    t2[i] = d[i] - c[i];
    b[i] = a[i] + t1[i];
    a[i+1] = b[i] * t2[i];
}
```

Figure 38: Example of node splitting

Scalar Expansion

This transformation replaces a scalar in a loop by an array to eliminate dependences in the loop body and enable parallelization of the loop nest. If the scalar is used after the loop, compensation code must be added.

```
for(i=0; i<N; i++){
    c = b[i];
    a[i] = a[i] + c;
}
```

```
for(i=0;i<N; i++){
    tmp[i] = b[i];
    a[i] = a[i] + tmp[i];
}
c = tmp[N-1];
```

Figure 39: Example of scalar expansion

Array Contraction / Array Shrinking

This transformation is the reverse transformation of scalar expansion. It may be needed if scalar expansion generates too many memory requirements.

```
for(i=0; i<N;i++)
    for(j=0; j<N;j++) {
        t[i][j] = a[i][j] * 3;
        b[i][j] = t[i][j] + c[j];
}
```

```
for(i=0; i<N;i++)
    for(j=0; j<N;j++) {
        t[j] = a[i][j] * 3;
        b[i][j] = t[j] + c[j];
}
```

Figure 40: Example of array contraction

Scalar Replacement

This transformation replaces an invariant array reference in a loop by a scalar. This array element is loaded in a scalar before the inner loop and stored again after the inner loop, if it is modified. It can be used in conjunction with loop interchange.

```
for(i=0; i<N; i++)
    for(j=0; j<N;j++)
        a[i] = a[i] + b[i][j];
```

```
for(i=0;i<N; i++){
    tmp = a[i];
    for(j=0; j<N;j++)
        tmp = tmp + b[i][j];
    a[i] = tmp;
}
```

Figure 41: Example of scalar replacement

Reduction Recognition

This transformation allows to handle reductions in loops. A reduction is an operation that computes a scalar value from arrays. It can be a dot product, the sum or minimum of a vector for instance. The

goal is then to perform as many operations in parallel as possible. One way is to accumulate a vector register of partial results and then reduce it to a scalar with a sequential loop. Maximum parallelism is then achieved by reducing the vector register with a tree: pairs of elements are summed, then pairs of these results are summed, etc.

```
for(i=0; i<N;i++)
    s = s + a[i];
```

```
for(i=0; i<N; i=i+64)
    tmp[0:63] = tmp[0:63] + a[i:i+63];
for(i=0; i<64;i++)
    s = s + tmp[i];
```

Figure 42: Example of reduction recognition

Loop Pushing / Loop Embedding

This transformation replaces a call in a loop body by the loop in the called function. It is an inter-procedural optimization. It allows the parallelization of the loop nest and eliminates the overhead caused by the procedure call. Loop distribution can be used in conjunction with loop pushing.

```
for(i=0; i<N; i++)
    f(x,i);

void f(int* a,int j) {
   a[j] = a[j] + c;
}
```

```
f2(x)

void f2(int* a) {
   for(i=0; i<N; i++)
       a[i] = a[i] + c;
}
```

Figure 43: Example of loop pushing

Procedure Inlining

This transformation replaces a call to a procedure by the code of the procedure itself. It is an inter-procedural optimization. It allows a loop nest to be parallelized, removes overhead caused by the procedure call, and can improve locality.

```
for(i=0; i<N; i++)
    f(a,i);

void f(int* x, int j){
    x[j] = x[j] + c;
}
```

```
for(i=0; i<N; i++)
    a[i] = a[i] + c;
```

Figure 44: Example of procedure inlining

Statement Reordering

This transformation schedules instructions of the loop body to modify the data dependence graph and enable vectorization.

```
for(i=0;i < N;i++) {
    a[i] = b[i] * 2;
    c[i] = a[i-1] - 4;
}
```

```
for(i=0; i<N; i++) {
    c[i] = a[i-1] - 4;
    a[i] = b[i] * 2;
}
```

Figure 45: Example of statement reordering

Software Pipelining

This transformation parallelizes a loop body by scheduling instructions of different instances of the loop body. It is a powerful optimization to improve instruction-level parallelism. It can be used in conjunction with loop unrolling. In the example below, the preload commands can be issued one after another, each taking only one cycle. This time is just enough to request the memory areas. It is not enough to actually load them. This takes many cycles, depending on the cache level that actually has the data. Execution of a configuration behaves similarly. The configuration is issued in a single cycle, waiting until all data are present. Then the configuration executes for many cycles. Software pipelining overlaps the execution of a configuration with the preloads for the next configuration. This way, the XPP array can be kept busy in parallel to the Load/Store unit.

```
Issue Cycle Command

            XPPPreloadConfig(CFG1);
            for (i=0; i<100; ++i) {
         1:   XPPPreload(2,a+10*i,10);
         2:   XPPPreload(5,b+20*i,20);
         3:
         4:   // delay
         5:
         6:   XPPExecute(CFG1);
            }

Issue Cycle Command
   Prologue XPPPreloadConfig(CFG1);
            XPPPreload(2,a,10);
            XPPPreload(5,b,20);
            // delay
            for (i=1; i<100; ++i) {
  Kernel 1:   XPPExecute(CFG1);
         2:   XPPPreload(2,a+10*i,10);
         3:   XPPPreload(5,b+20*i,20);
         4: }
            XPPExecute(CFG1);
     Epilog // delay
```

Figure 46: Example of software pipelining

Vector Statement Generation

This transformation replaces instructions by vector instructions that can perform an operation on several data in parallel.

```
for(i=0; i<=N; i++)          a[0:N] = b[0:N];
   a[i] = b[i];
```

Figure 47: Example of vector statement generation

3.2.3 Data-Layout Optimizations

In the following we describe optimizations that modify the data layout in memory in order to extract more parallelism or prevent memory problems like cache misses.

Scalar Privatization

This optimization is used in multi-processor systems to increase the amount of parallelism and avoid unnecessary communications between the processing elements. If a scalar is only used like a temporary variable in a loop body, then each processing element can receive a copy of it and achieve its computations with this private copy.

```
for(i=0;i <= N;i++) {
    c = b[i];
    a[i] = a[i] + c;
}
```

Figure 48: Example for scalar privatization

Array Privatization

This optimization is the same as scalar privatization except that it works on arrays rather than on scalars.

Array Merging

This optimization transforms the data layout of arrays by merging the data of several arrays following the way they are accessed in a loop nest. This way, memory cache misses can be avoided. The layout of the arrays can be different for each loop nest. Below is the example of a cross-filter, where the accesses to array *a* are interleaved with accesses to array *b*. The picture next to it represents the data layout of both arrays where blocks of *a* (in green) are merged with blocks of *b* (in yellow). Unused memory space is in white. Thus cache misses are avoided as data blocks containing arrays *a* and *b* are loaded into the cache when getting data from memory. More details can be found in [11].

```
for(j=1;j<=N-1;i++)
   for(j=1;j<=N;j++)
      b[i][j] = 0.25*(a[i-1][j] + a[i][j-1] +
a[i+1][j] + a[i][j+1]);
```

Figure 49: Example for array merging

3.2.4 Example of application of the optimizations

As seen before a lot of optimizations can be performed on loops before and also after generation of vector statements. Finding a sequence of optimizations that would produce an optimal solution for all loop nests of a program is still an area of research. Therefore we can only propose a way to use these optimizations that follows a reasonable heuristic to produce vectorizable loop nests. To vectorize the code, we can use the Allen-Kennedy algorithm that uses statement reordering and loop distribution before vector statements are generated. It can be enhanced with loop interchange, scalar expansion, index set splitting, node splitting, loop peeling. All these transformations are based on the data dependence graph. A statement can be vectorized if it is not part of a dependence cycle, hence optimizations are performed to break cycles or, if not completely possible, to create loop nests without dependence cycles.

We can divide the whole process in four majors steps. First we should restructure the procedures by analyzing the procedure calls inside the loop bodies and try to remove them. Then some high-level dataflow optimizations are applied to the loop bodies to modify their control-flow and simplify their code. The third step would consist in preparing the loop nests for vectorization by building perfect

loop nests and ensuring that inner loop levels are vectorizable. Then optimizations can be performed that target the architecture and optimize the data locality. It should also be noted that other optimizations and code transformations can occur between these different steps that can also help to further optimize the loop nests.

Hence the first step applies procedure inlining and loop pushing to remove the procedure calls of the loop bodies. Then the second step consists of loop-invariant code motion, loop unswitching, strength reduction and idiom recognition. The third step can be divided in several subsets of optimizations. We can first apply loop reversal, loop normalization and if-conversion to get normalized loop nests. This allows to build the data dependency graph. Then if dependences prevent the loop nest to be vectorized transformations can be applied. For instance if dependences occur only on certain iterations, loop peeling or loop splitting can be applied. Node splitting, loop skewing, scalar expansion or statement reordering can be applied in other cases. Then loop interchange moves inwards the loop levels without dependence cycles. The goal is to have perfectly nested loops with the loop levels carrying dependence cycles as much outwards as possible. Then we can apply loop fusion, reduction recognition, scalar replacement/array contraction and loop distribution to further improve the following vectorization. Vector statement generation can be performed at last using the Allen-Kennedy algorithm for instance. The last step can consist of optimizations like loop tiling, strip-mining, loop unrolling and software pipelining that take into account the target processor.

The number of optimizations in the third step is large, but not all of them are applied to each loop nest. Following the goal of the vectorization and the data dependence graph only some of them are applied. Heuristics are used to guide the application of the optimizations, that can be applied several times if needed. Let us illustrate this with an example.

```
void f(int** a, int** b, int *c,int i, int j) {
     a[i][j] = a[i][j-1] - b[i+1][j-1];
}

void g(int* a,int* c,int i) {
      a[i] = c[i] + 2;
}


for(i=0; i<N;i++) {
   for(j=1; j<9;j=j++)
      if (k>0)
         f(a,b,i,j);
      else
         g(d,c,j);
    }
  d[i] = d[i+1] + 2;
}

for(i=0; i<N;i++)
    a[i][i] = b[i] + 3;
```

The first step will find that inlining the two procedure calls is possible, then loop unswitching can be applied to remove the conditional instruction of the loop body. The second step begins by applying loop normalization and analyses the data dependence graph. A cycle can be broken by applying loop interchange as it is only carried by the second level. The two levels are exchanged, so that the inner level is vectorizable. Before that or also after, we apply loop distribution. Loop fusion can be applied when the loop on i is pulled out of the conditional instruction by a traditional redundant code elimination optimization. Finally vector code can be generated for the resulting loops.

So in more details, after procedure inlining, we obtain:

```
for(i=0; i<N;i++) {
   for(j=1; j<9;j=j++)
      if (k>0)
         a[i][j] = a[i][j-1] - b[i+1][j-1];
      else
         d[j] = c[j] + 2;
    }
   d[i] = d[i+1] + 2;
}

for(i=0; i<N;i++)
    a[i][i] = b[i] + 3;
```

After loop unswitching, we obtain:

```
if (k > 0)
  for(i=0; i<N;i++) {
      for(j=1; j<9;j=j++)
         a[i][j] = a[i][j-1] - b[i+1][j-1];
      d[i] = d[i+1] + 2;
     }
else
  for(i=0; i<N;i++) {
      for(j=1; j<9;j=j++)
         d[j] = c[j] + 2;
      d[i] = d[i+1] + 2;
}

for(i=0; i<N;i++)
    a[i][i] = b[i] + 3;
```

After loop normalization, we obtain:

```
if (k > 0)
  for(i=0; i<N;i++) {
      for(j=0; j<8;j=j++)
         a[i][j+1] = a[i][j] - b[i+1][j];
      d[i] = d[i+1] + 2;
   }
else
  for(i=0; i<N;i++) {
      for(j=0; j<8;j=j++)
         d[j] = c[j+1] + 2;
      d[i] = d[i+1] + 2;
}

for(i=0; i<N;i++)
    a[i][i] = b[i] + 3;
```

After loop distribution and loop fusion, we obtain:

```
if (k > 0)
  for(i=0; i<N;i++)
      for(j=0; j<8;j=j++)
         a[i][j+1] = a[i][j] - b[i+1][j];
else
  for(i=0; i<N;i++)
      for(j=0; j<8;j=j++)
         d[j] = c[j+1] + 2;

for(i=0; i<N;i++) {
    d[i] = d[i+1] + 2;
    a[i][i] = b[i] + 3;
}
```

After loop interchange, we obtain:

```
if (k > 0)
   for(j=0; j<8;j=j++)
       for(i=0; i<N;i++)
         a[i][j+1] = a[i][j] - b[i+1][j];
else
  for(i=0; i<N;i++)
      for(j=0; j<8;j=j++)
          d[j] = c[j+1] + 2;

for(i=0; i<N;i++) {
    d[i] = d[i+1] + 2;
    a[i][i] = b[i] + 3;
}
```

After vector code generation, we obtain

```
if (k > 0)
   for(j=0; j<8;j=j++)
      a[0:N-1][j+1] = a[0:N-1][j] - b[0:N][j];
else
   for(i=0; i<N;i++)
      d[0:8] = c[1:9] + 2;

d[0:N-1] = d[1:N] + 2;
a[0:N-1][0:N-1] = b[0:N] + 3;
```

4 Compiler Specification for the PACT XPP

4.1 Introduction

A cached RISC-XPP architecture exploits its full potential on code that is characterized by high data locality and high computational effort. A compiler for this architecture has to consider these design constraints. The compiler's primary objective is to concentrate computational expensive calculations to innermost loops and to make up as much data locality as possible for them.

The compiler contains usual analysis and optimizations. As interprocedural analysis, like alias analysis, are especially useful, a global optimization driver is necessary to ensure the propagation of global information to all optimizations. The following sections concentrate on the way the PACT XPP influences the compiler.

4.2 Compiler Structure

Figure 50 shows the main steps the compiler must follow to produce code for a system containing a RISC processor and a PACT XPP. The next sections focus on the XPP compiler itself, but first the other steps are briefly described.

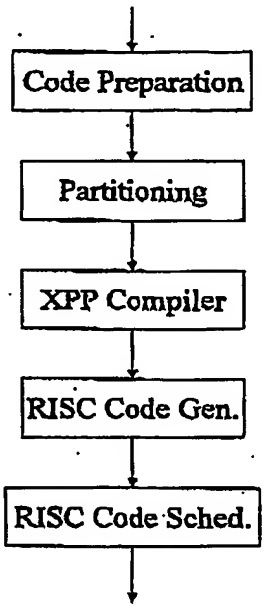


Figure 50: Global View of the Compiling Process

4.2.1 Code Preparation

This step takes the whole program as input and can be considered as a usual compiler front-end. It will prepare the code by applying code analysis and optimizations to enable the compiler to extract as many loop nests as possible to be executed by the PACT XPP. Important optimizations are idiom recognition, copy propagation, dead code elimination, and all usual analysis like dataflow and alias analysis.

4.2.2 Partitioning

Partitioning decides which part of the program is executed by the host processor and which part is executed by the PACT XPP.

A loop nest is executed by the host in three cases:

- if the loop nest is not well-formed,
- if the number of operations to execute is not worth it to be executed on the PACT XPP, or
- if it is impossible to get a mapping of the loop nest on the PACT XPP.

A loop nest is said to be well-formed if the loop bounds and the step of all loops are constant, the loop induction variables are known and if there is only one entry and one exit to the loop nest.

Another problem arises with loop nests where the loop bounds are constant but unknown at compile time. Loop tiling allows to overcome this problem, it will be described below. Nevertheless it could be that it is not worth it to execute the loop nest on the PACT XPP if the loop bounds are too low. A conditional instruction testing if the loop bounds are large enough can be introduced, and 2 versions of the loop nest are produced. One would be executed on the host processor, and the other on the PACT

XPP when the loop bounds are suitable. This would also ease applications of loop transformations, as possible compensation code would be simpler due to the hypothesis on the loop bounds.

4.2.3 RISC Code Generation and Scheduling

After the XPP compiler has produced NML code for the loops chosen by the partitioning phase, the main compiling process must handle the code that will be executed by the host processor where instructions to manage the configurations have been inserted. This is the aim of the last two steps:

- RISC Code Generation and
- RISC Code Scheduling.

The first one produces code for the host processor and the second one optimizes it further by looking for a better scheduling using software pipelining for instance.

4.3 XPP Compiler for Loops

Figure 51 describes the internal processing of the XPP Compiler. It is a complex cooperation between program transformations, included in the XPP Loop Optimizations, a temporal partitioning phase, NML code generation and the mapping of the configuration on the PACT XPP.

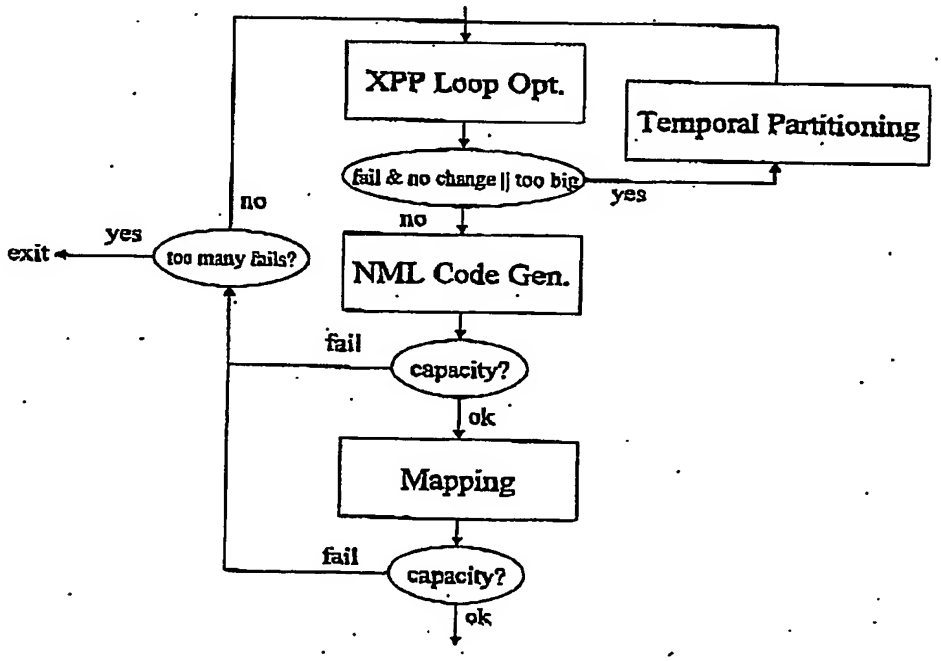


Figure 51: Detailed Architecture of the XPP Compiler

First loop optimizations targeted at the PACT XPP are applied to try to produce innermost loop bodies that can be executed on the array of processors. If this is the case, the NML code generation phase is

called, if not then temporal partitioning is applied to get several configurations for the same loop. After NML code generation and the mapping phase, it can also happen that a configuration will not fit on the PACT XPP. In this case the loop optimizations are applied again with respect to the reasons of failure of the NML code generation or of the mapping. If this new application of loop optimizations does not change the code, temporal partitioning is applied. Furthermore we keep track of the number of attempts for the NML Code Generation and the mapping, if too many attempts are made, and we still do not obtain a solution, we break the process, and the loop nest will be executed by the host processor.

4.3.1 Temporal Partitioning

Temporal partitioning splits the code generated for the PACT XPP in several configurations if the number of operations, i.e. the size of the configuration, to be executed in a loop nest exceeds the number of operations executable in a single configuration. This transformation is called loop dissevering [6]. These configurations are then integrated in a loop of configurations whose number of execution corresponds to the iteration range of the original loop.

4.3.2 Generation of NML Code

This step takes as input an intermediate form of the code produced by the XPP Loop Optimizations step, together with a dataflow graph built upon it. NML code can then be produced by using tree- or DAG-pattern matching techniques.

4.3.3 Mapping Step

This step takes care of mapping the NML modules on the PACT XPP by placing the operations on the ALUs, FREGs, and BREGs, and routing the data through the buses.

4.4 XPP Loop Optimizations Driver

The loop optimizations used for the PACT XPP are now described. Their goal is to extract as much parallelism as possible from the loop nests in order to execute them on the PACT XPP by exploiting the ALU-PAEs as effectively as possible and to avoid memory bottlenecks with the IRAMs. The following sections explain how they are organized and how to take into account the architecture for applying the optimizations.

4.4.1 Organization of the System

Figure 52 below presents the organization of the loop optimizations. The transformations are divided in six groups. Other standard optimizations and analysis are applied in-between. Each group could be called several times. Loops over several groups can also occur if needed. The number of iterations for each driver loop can be of constant value or determined at compile time by the optimizations itself (e.g. repeat until a certain code quality is reached). In the first iteration of the loop, it can be checked if loop nests are usable for the PACT XPP, it is mainly directed to check the loop bounds etc. For instance if the loop nest is well-formed and the data dependence graph does not prevent optimization, but the loop bounds are unknown, then in the first iteration loop tiling is applied to get an innermost that is easier to handle and can be better optimized, and in the second iteration, loop normalization, if-conversion, loop interchange and other optimizations can be applied to effectively optimize the

innermost loops for the PACT XPP. Nevertheless this has not been necessary until now with the examples presented in the next chapters.

Group I ensures that no procedure calls occur in the loop nest. Group II prepares the loop bodies by removing loop-invariant instructions and conditional instruction to ease the analysis. Group III generates loop nests suitable for the data dependence analysis. Group IV contains optimizations to transform the loop nests to get data dependence graphs that are suitable for vectorization. Group V contains optimizations that ensure that the innermost loops can be executed on the PACT XPP. Group VI contains optimizations that further extract parallelism from the loop bodies. Group VII contains optimizations more towards optimizing the usage of the hardware itself.

In each group the application of the optimizations depends on the result of the analysis and the characteristics of the loop nest. For instance it is clear that not all transformations in Group IV are applied. It depends on the data dependence graph computed before.

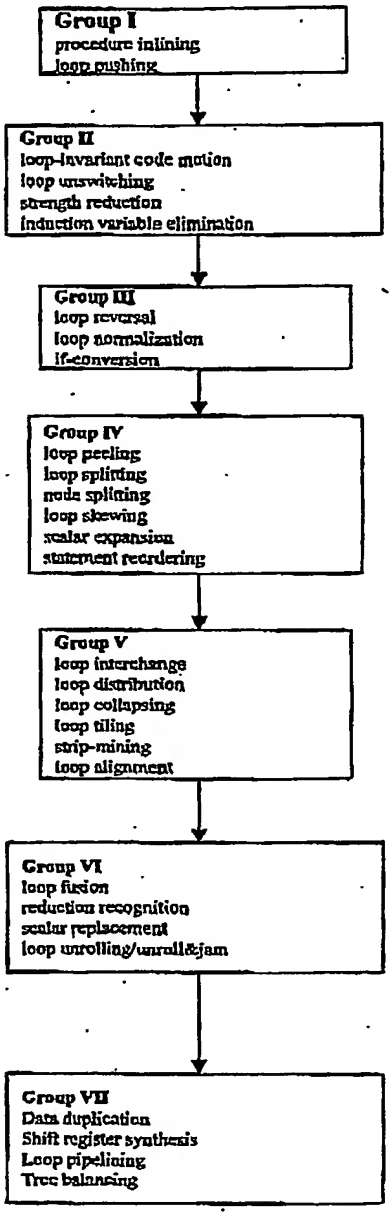


Figure 52:Detailed View of the XPP Loop Optimizations

4.4.2 Loop Preparation

The optimizations of Groups I, II and III of the XPP compiler generate loop bodies without procedure calls, conditional instructions and induction variables other than loop control variables. Thus loop nests, where the innermost loops are suitable for execution on the PACT XPP, are obtained. The iteration ranges are normalized to ease data dependence analysis and the application of other code transformations.

4.4.3 Transformation of the Data Dependence Graph

The optimizations of Group IV are performed to obtain innermost loops suitable for vectorization with respect to the data dependence graph. Nevertheless a difference with usual vectorization is that a dependence cycle, that would normally prevent any vectorization of the code, does not prevent the optimization of a loop nest for the PACT XPP. If a cycle is due to an anti-dependence, then it could be that it won't prevent optimization of the code as stated in [7]. Furthermore dependence cycles will not prevent vectorization for the PACT XPP when it consists only of a loop-carried true dependence on the same expression. If cycles with distance k occur in the data dependence graph, then this can be handled by holding k values in registers. This optimization is of the same class as cycle shrinking.

Nevertheless limitations due to the dependence graph exist. Loop nests cannot be handled if some dependence distances are not constant, or unknown. If only a few dependences prevent the optimization of the whole loop nest, this could be overcome, by using the traditional vectorization algorithm that sorts topologically the strongly connected components of the data dependence graph (statement reordering), and then apply loop distribution. This way, loop nests, which can be handled by the PACT XPP and some by the host processor, can be obtained.

4.4.4 Influence of the Architectural Parameters

Some hardware specific parameters influence the application of the loop transformations. The number of operations and memory accesses, that a loop body performs, is estimated at each step. These parameters influence loop unrolling, strip-mining, loop tiling and also loop interchange (iteration range).

The table below lists the parameters that influence the application of the optimizations. For each of them two data are given: a starting value computed from the loop, and a restriction value which is the value the parameter should reach or should not exceed after the application of the optimizations. Vector length depicts the range of the innermost loops, i.e. the number of elements of an array accessed in the loop body. Reused data set size represents the amount of data that must fit in the cache. I/O IRAMs, ALU, FREG, BREG stand for the number of IRAMs, ALUs, FREGs, and BREGs respectively that constitute the PACT XPP. The dataflow graph width represents the number of operations that can be executed in parallel in the same pipeline stage. The dataflow graph height represents the length of the pipeline. Configuration cycles amounts to the length of the pipeline, and to the number of cycles dedicated to the control. The application of each optimization may

- decrease a parameter's value (-),
- increase a parameter's value (+),
- not influence a parameter (id), or
- adapt a parameter's value to fit into the goal size (make fit).

Furthermore, some resources must be kept for control in the configuration; this means that the optimizations should not make the needs exceed more than 70-80% each resource.

Parameter	Goal	Starting Value
Vector length	IRAM size (256 words)	Loop count
Reused data set size	Approx. cache size	Algorithm analysis/loop sizes
I/O IRAMs	PACT size (16)	Algorithm inputs + outputs
ALU	PACT size (< 64)	ALU opcode estimate
BREG	PACT size (< 80)	BREG opcode estimate
FREG	PACT size (< 80)	FREG opcode estimate
Data flow graph width	High	Algorithm data flow graph
Data flow graph height	Small	Algorithm data flow graph
Configuration cycles	≤ command line parameter	Algorithm analysis

Here are some additional notations used in the following descriptions. Let *n* be the total number of processing elements available, *r*, the width of the dataflow graph, *in*, the maximum number of input values in a cycle and *out*, the maximum number of output values possible in a cycle. On the PACT XPP, *n* is the number of ALUs, FREGs and BREGs available for a configuration, *r* is the number of ALUs, FREGs and BREGs that can be started in parallel in the same pipeline stage and, *in* and *out* amount to the number of available IRAMs. As IRAMs have 1 input port and 1 output port, the number of IRAMs yields directly the number of input and output data.

The number of operations of a loop body is computed by adding all logic and arithmetic operations occurring in the instructions. The number of input values is the number of operands of the instructions regardless of address operations. The number of output values is the number of output operands of the instructions regardless of address operations. To determine the number of parallel operations, input and output values, and the dataflow graph must be considered. The effects of each transformation on the architectural parameters are now presented in detail.

Loop Interchange

Loop interchange is applied when the innermost loop has a too narrow iteration range. In that case, loop interchange allows to have an innermost loop with a more profitable iteration range. It can also be influenced by the layout of the data in memory. It can be profitable to data locality to interchange two loops to get a more practical way to access arrays in the cache and therefore prevent cache misses. It is of course also influenced by data dependences as explained earlier.

Parameter	Effect
Vector length	+
Reused data set size	make fit
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	id
Data flow graph height	id
Configuration cycles	-

Loop Distribution

Loop distribution is applied if a loop body is too big to fit on the PACT XPP. Its main effect is to reduce the processing elements needed by the configuration. Reducing the need for IRAMs can only be a side effect.

Parameter	Effect
Vector length	id
Reused data set size	id
I/O IRAMs	make fit
ALU	make fit
BREG	make fit
FREG	make fit
Data flow graph width	-
Data flow graph height	-
Configuration cycles	-

Loop Collapsing

Loop collapsing can be used to make the loop body use more memory resources. As several dimensions are merged, the iteration range is increased and the memory needed in increased as well.

Parameter	Effect
Vector length	+
Reused data set size	+
I/O IRAMs	+
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	+
Configuration cycles	+

Loop Tiling

Loop tiling, as multi-dimensional strip-mining, is influenced by all parameters, it is especially useful when the iteration space is by far too big to fit in the IRAM, or to guarantee maximum execution time when the iteration space is unbounded (see Section 4.4.6). It can then make the loop body fit with respect to the resources of the PACT XPP, namely the IRAM and cache line sizes. The size of the tiles for strip-mining and loop tiling can be computed like this:

tile size = resources available for the loop body / resources necessary for the loop body

The resources available for the loop body are the whole resources of the PACT XPP for this configuration. A tile size can be computed for the data and another one for the processing elements, the final tile size is then the minimum between these two. For instance, when the amount of data accessed is larger than the capacity of the cache, loop tiling can be applied like below.

```
for(i=0;i <= 1048576;i++)
    <loop body>
```

```
for(i=0; i<= 1048576; i+= CACHE_SIZE)
    for(j=0; j< CACHE_SIZE; j+=IRAM_SIZE)
        for(k=0; k<IRAM_SIZE;k++)
         <tiled loop body>
```

Figure 53: Example of loop tiling for the PACT XPP

Parameter	Effect
Vector length	make fit
Reused data set size	make fit
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	+
Configuration cycles	+

Strip-Mining

Strip-mining is used to make the amount of memory accesses of the innermost loop fit with the IRAMs capacity. The processing elements do not usually represent a problem as the PACT XPP has 64 ALU-PAEs which should be sufficient to execute any single loop body. Nevertheless, the number of operations can be also taken into account the same way as the data.

Parameter	Effect
Vector length	make fit
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	-

Loop Fusion

Loop fusion is applied when a loop body does not use enough resources. In this case several loop bodies can be merged to obtain a configuration using a larger part of the available resources.

Parameter	Effect
Vector length	id
Reused data set size	id
I/O IRAMs	+
ALU	+
BREG	+
FREG	+
Data flow graph width	id
Data flow graph height	+
Configuration cycles	+

Scalar Replacement

The amount of memory needed by the loop body should always fit in the IRAMs. Thanks to this optimization, some input or output data represented by array references, that should be stored in IRAMs, are replaced by scalars, that are either stored in FREGs or kept on buses.

Parameter	Effect
Vector length	+
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	id

Loop Unrolling

Loop unrolling, loop collapsing, loop fusion and loop distribution are influenced by the number of operations of the body of the loop nest and the number of data inputs and outputs of these operations, as they modify the size of the loop body. The number of operations should always be smaller than n, and the number of input and output data should always be smaller than in and out.

Parameter	Effect
Vector length	id
Reused data set size	id
I/O IRAMs	+
ALU	+
BREG	+
FREG	+
Data flow graph width	id
Data flow graph height	+
Configuration cycles	+

Unroll-and-Jam

Unroll-and-Jam consists in unrolling an outer loop and then merging the inner loops. It must compute the unrolling degree u with respect to the number of input memory accesses m and output memory accesses p in the inner loop. The following inequality must hold: $u * m \leq in \wedge u * p \leq out$. Moreover the number of operations of the new inner loop must also fit on the PACT XPP.

Parameter	Effect
Vector length	id
Reused data set size	+
I/O IRAMs	+
ALU	+
BREG	+
FREG	+
Data flow graph width	id
Data flow graph height	+
Configuration cycles	+

4.4.5 Optimizations Towards Hardware Improvements

At this step other optimizations, specific to the PACT XPP, can be made. These optimizations deal mostly with memory problems and dataflow considerations. This is the case of shift register synthesis, input data duplication (similar to scalar privatization), or loop pipelining.

Shift Register Synthesis

This optimization deals with array accesses that occur during the execution of a loop body. When several values of an array are alive for different iterations, it can be convenient to store them in registers rather than accessing memory each time they are needed. As the same value must be stored in

different registers depending on the number of iterations it is alive, a value shares several registers and flows from a register to another at each iteration. It is similar to a vector register allocated to an array access with the same value for each element. This optimization is performed directly on the dataflow graph by inserting nodes representing registers when a value must be stored in a register. In the PACT XPP, it amounts to store it in a data register. A detailed explanation can be found in [1].

Shift register synthesis is mainly suitable for small to medium amounts of iterations where values are alive. Since the pipeline length increases with each iteration for which the value has to be buffered, the following method is better suited for medium to large distances between accesses in one input array.

Nevertheless this method works very well for image processing algorithms which mostly alter a pixel by analyzing itself and its surrounding neighbors.

Parameter	Effect
Vector length	+
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	id

Input Data Duplication

This optimization is orthogonal to shift register synthesis. If different elements of the same array are needed concurrently, instead of storing the values in registers, the same values are copied in different IRAMs. The advantage against shift register synthesis is the shorter pipeline length, and therefore the increased parallelism, and the unrestricted applicability. On the other hand, the cache-IRAM bottleneck can affect the performance of this solution, depending on the amounts of data to be moved. Nevertheless we assume that cache-IRAM transfers are negligible to transfers in the rest of the memory hierarchy.

Parameter	Effect
Vector length	+
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	id

Loop Pipelining

This optimization consists in synchronizing operations by inserting delays in the dataflow graph. These delays are registers. For the PACT XPP, it amounts to store values in data registers to delay the operation using them. This is the same as pipeline balancing performed by xmap.

Parameter	Effect
Vector length	+
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	+

Tree Balancing

This optimization consists in balancing the tree representing the loop body. It reduces the depth of the pipeline, thus reducing the execution time of an iteration, and increases parallelism.

Parameter	Effect
Vector length	+
Reused data set size	id
I/O IRAMs	id
ALU	id
BREG	id
FREG	id
Data flow graph width	+
Data flow graph height	-
Configuration cycles	-

4.4.6 Limiting the Execution Time of a Configuration

The execution time of a configuration must be controlled. This is ensured in the compiler by strip-mining and loop tiling that take care that not more input data as the IRAMs capacity come in the PACT XPP in a cycle. This way the iteration range of the innermost loop that is executed on the PACT XPP is limited, and therefore its execution time. Moreover partitioning ensures that loops, whose execution count can be computed at run time, are going to be executed on the PACT XPP. This condition is trivial for for-loops, but for while-loops, where the execution count cannot de determined statically, a transformation like sketched below can be applied. As a result, the inner for-loop can be handled by the PACT XPP.

```
while (ok) {
   <loop body>
}
```

```
while (ok)
  for(i=0; i<100 && ok; i++) {
      <loop body>
  }
```

Figure 54: Transformation of while-loops

5 Case Studies

5.1 3x3 Edge Detector

5.1.1 Original Code

Source Code:

```
#define VERLEN 16
#define HORLEN 16
main() {
  int v, h, inp;
  int p1[VERLEN][HORLEN];
  int p2[VERLEN][HORLEN];
  int htmp, vtmp, sum;

  for(v=0; v<VERLEN; v++)          // loop nest 1
       for(h=0; h<HORLEN; h++) {
            scanf("%d", &p1[v][h]); // read input pixels to p1
            p2[v][h] = 0; // initialize p2
       }

  for(v=0; v<=VERLEN-3; v++) {  // loop nest 2
    for(h=0; h<=HORLEN-3; h++) {
      htmp = (p1[v+2][h] - p1[v][h]) +
             (p1[v+2][h+2] - p1[v][h+2]) +
             2 * (p1[v+2][h+1] - p1[v][h+1]);
      if (htmp < 0)
        htmp = - htmp;

      vtmp = (p1[v][h+2] - p1[v][h]) +
             (p1[v+2][h+2] - p1[v+2][h]) +
             2 * (p1[v+1][h+2] - p1[v+1][h]);
      if (vtmp < 0)
        vtmp = - vtmp;

      sum = htmp + vtmp;
      if (sum > 255)
        sum = 255;
      p2[v+1][h+1] = sum;
    }
  }
  for(v=0; v<VERLEN; v++)        // loop nest 3
    for(h=0; h<HORLEN; h++)
      printf("%d\n", p2[v][h]); // print output pixels from p2
}
```

5.1.2 Preliminary Transformations

Interprocedural Optimizations

The first step normally invokes interprocedural transformations like function inlining and loop pushing. Since no procedure calls are within the loop body, these transformations are not applied to this example.

Partitioning

The partitioning algorithm chooses which code runs on the RISC processor and which code runs on the XPP. Since we only consider inner loops to run on the XPP, the basic blocks are annotated with the loop nest depth. Thus basic blocks which are not in a loop are separated out. Furthermore function calls within a loop body prevent a loop to be considered for running on the XPP.

In our benchmark the loop nests 1 and 3 are marked as to run on the RISC host because of the function call. In the following sections they are not considered any further.

It is to say that at this compilation stage it is not predictable if the remaining loop nests can be synthesized for the XPP. We just separated the ones which definitely cannot run on it, others may follow, since running the code on the RISC CPU is always the reassurance in our strategy.

Loop Analysis and Normalization

The code upon has already normalized loops. Nevertheless it is more likely that human written code would look like

```
for(v=1; v < VERLEN - 1; v++) {
  for(h=1; h < HORLEN - 1; h++) {
      htmp = (p1[v+1][h-1] - p1[v-1][h-1]) +
             (p1[v+1][h+1] - p1[v-1][h+1]) +
             2 * (p1[v+1][h] - p1[v-1][h]);
      if (htmp < 0)
        htmp = - htmp;

      vtmp = (p1[v-1][h+1] - p1[v-1][h-1]) +
             (p1[v+1][h+1] - p1[v+1][h-1]) +
             2 * (p1[v][h+1] - p1[v][h-1]);
      if (vtmp < 0)
        vtmp = - vtmp;

      sum = htmp + vtmp;
      if (sum > 255)
        sum = 255;
      p2[v+1][h+1] = sum;
  }
}
```

Although seen at first sight by a human reader, it is not obvious for the compiler that the loop is well formed. Therefore it is tried to normalize the loop.

If the original loop induction variable is called i with the increment value s and lower and upper loop bounds l and u, respectively, then the normalized loop with the induction variable i' and the upper bound u' (the lower bound l' is 0 by definition) is transformed as follows:

- The upper bound calculates to u' = (u-l)/s.
- All occurrences of i are replaced by l + i' * s.

Applied to the code above, the loop statement `for(v=1; v < VERLEN - 1; v++)` with the lower bound vl = 1, the upper bound vu = 14 (< 15 means <= 14 in integer arithmetic) and the increment vs = 1 transforms to

```
for(vn=0; vn <= (vu - vl)/vs; vn++)
```

or simplified

```
for(vn=0; vn <= 13; vn++)
```

The 'h-loop' is transformed equally, issuing the original code.

Idiom Recognition

In the second step idiom recognition finds the abs() and min() structures in the loop body. Please note that although the XPP has no `abs` opcode, it can easily be synthesized and should therefore be produced to simplify the internal representation (otherwise if-conversion has to handle this case which increases the complexity).

Therefore the code after idiom recognition looks like (abs() and min() are compiler known functions which are directly mapped to XPP opcodes or predefined NML modules)

```
for(v=0; v<=16-3; v++) {
  for(h=0; h<=16-3; h++) {
      htmp = (p1[v+2][h] - p1[v][h]) +
             (p1[v+2][h+2] - p1[v][h+2]) +
             2 * (p1[v+2][h+1] - p1[v][h+1]);
      htmp = abs(htmp);

      vtmp = (p1[v][h+2] - p1[v][h]) +
             (p1[v+2][h+2] - p1[v+2][h]) +
             2 * (p1[v+1][h+2] - p1[v+1][h]);
      vtmp = abs(vtmp);

      sum = min(htmp + vtmp, 255);
      p2[v+1][h+1] = sum;
  }
}
```

Dependency Analysis

```
        for(v=0; v<=16-3; v++) {

          for(h=0; h<=16-3; h++) {

S1            htmp = (p1[v+2][h] - p1[v][h]) +
                     (p1[v+2][h+2] - p1[v][h+2]) +
                     2 * (p1[v+2][h+1] - p1[v][h+1]);
S2            htmp = abs(htmp);

S3            vtmp = (p1[v][h+2] - p1[v][h]) +
                     (p1[v+2][h+2] - p1[v+2][h]) +
                     2 * (p1[v+1][h+2] - p1[v+1][h]);
S4            vtmp = abs(vtmp);
```

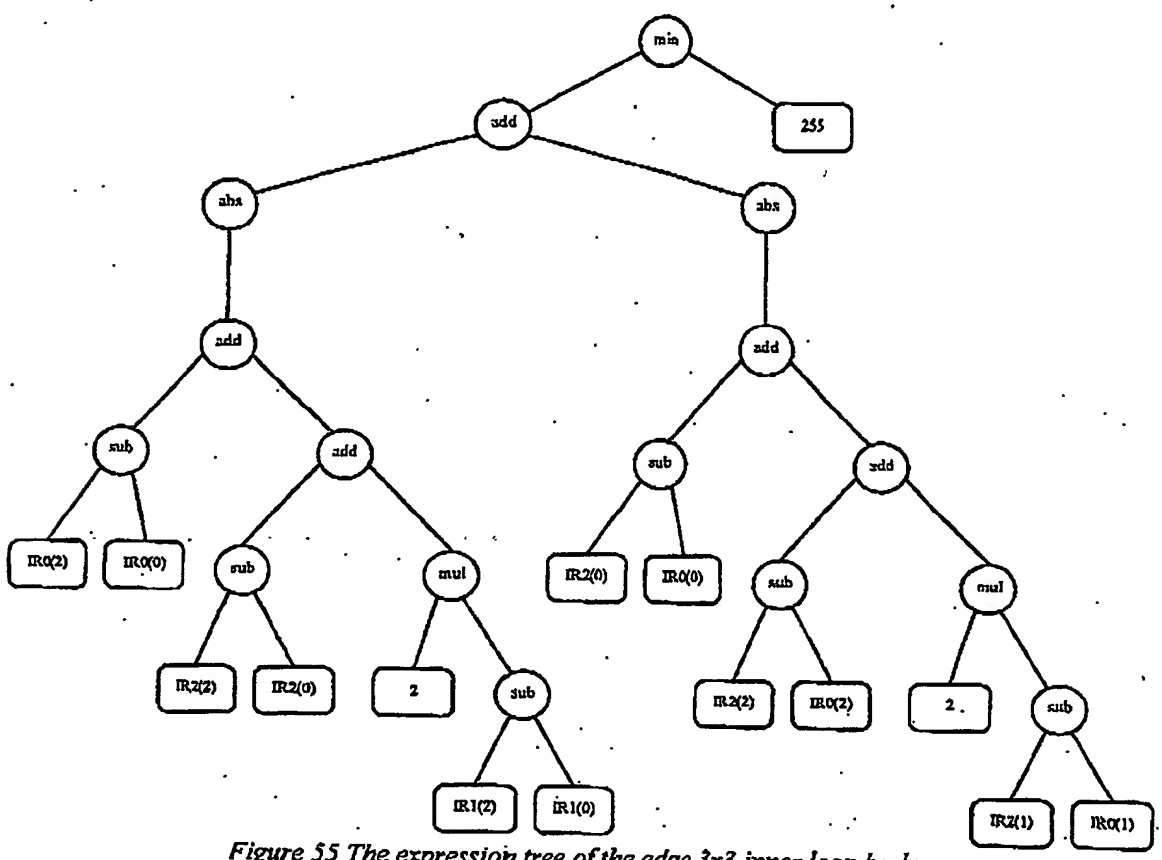


Figure 55 The expression tree of the edge 3x3 inner loop body

```
S5              sum = min(htmp + vtmp, 255);
S6              p2[v+1][h+1] = sum;
            }
        }
```

There are no loop carried dependencies which prevent pipeline vectorization. The loop independent scalar dependencies do not prevent pipeline vectorization since the transformation does not disturb the order of reads and writes. Furthermore forward expression substitution / dead code elimination will remove the scalars completely.

5.1.3 Pre Code Generation Transformations

Forward Expression Substitution / Dead Code Elimination

The lack of uses of htmp, vtmp and sum after the loop nest allows forward expression substitution along with dead code elimination to place the whole calculation into one statement.

```
p2[v+1][h+1] = min(abs(  (p1[v+2][h] - p1[v][h]) +
                         (p1[v+2][h+2] p1[v][h+2]) +
                         2 * (p1[v+2][h+1] - p1[v][h+1]))
               +  abs(   (p1[v][h+2] - p1[v][h]) +
                         (p1[v+2][h+2] - p1[v+2][h]) +
                         2 * (p1[v+1][h+2] - p1[v+1][h])), 255);
```

The scalar accesses then disappear completely.

Mapping to IRAMs

The array accesses are mapped to IRAMs. At this stage the IRAM numbers are chosen arbitrarily, the actual mapping to XPP IRAMs is done later.

Therefore we rename p1[v+x][h+y] and p2[v+x][h+y] to iramN[y]) (e.g. p1[v+2][h] to iram2[0]). The code reads then

```
iram3[1] = min(abs(iram2[0] - iram0[0])+
               (iram2[2] - iram0[2]) +
           2 * (iram2[1] - iram0[1]) +
           abs(iram0[2] - iram0[0])  +
               (iram2[2] - iram2[0]  +
           2 * (iram1[2] - iram1[0]), 255);
```

Tree Balancing

The visualized expression tree in Figure 55 shows another valuable optimization before matching the tree. Since the depth of the tree determines the length of the synthesized pipeline, another simplification can decrease this depth. In both of the main sub trees the operands of the commutative add expressions can be interchanged to reduce the overall tree depth.

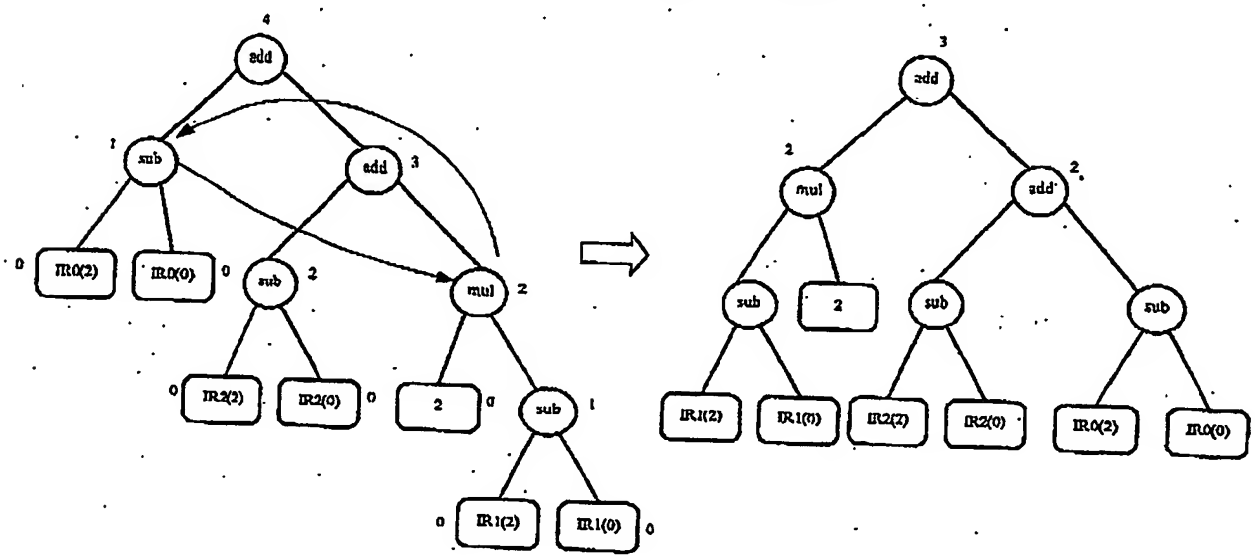


Figure 56 One of the sub trees before and after balancing. The numbers represent the annotated maximum tree depth from the node to its deepest child leaf node

The resulting expression tree is shown in Figure 56.

5.1.4 XPP Code generation

Pipeline Synthesis

As already stated the pipeline is synthesized by a dynamic programming tree matcher. In contrast to sequential processors it does not generate instructions and register references but PAE opcodes and port connections. The main calculation network is shown in Figure 57. The input data preparation network is not shown in this figure. The case of synthesized shift registers are shown in Figure 58,

while the variant with duplicated input data simply consists of an IRAM for each input channel in Figure 57.

Although this is straight forward, there remains the question how to access the different offsets of the vector register accesses. Although the RAM-PAEs are dual ported it is obvious that it is not possible to read different addresses concurrently.

Since it is not efficient to synthesize a configuration which generates the different addresses sequentially and demultiplexes the read operands into different branches of the data flow, other arrangements have to be made.

The two possibilities to access input data presented in subsection 4.4.5 yield the following in RISC pseudo code and XPP utilization.. The pseudo code running on the RISC core looks like

```
XPPPreload(config)
for(v=0; v<=16-3; v++) {
      XPPPreload(0, &p1[v], 16)
      XPPPreload(1, &p1[v+1], 16)
      XPPPreload(2, &p1[v+2], 16)
      XPPPreloadClean(3, &p2[v+1], 16)
      XPPExecute(config, IRAM(0), IRAM(1), IRAM(2), IRAM(3))
}
```

for shift register synthesis and like

```
XPPPreload(config)
for(v=0; v<=16-3; v++) {
      XPPPreload(0, &p1[v], 16)
      XPPPreload(1, &p1[v], 16)
      XPPPreload(2, &p1[v], 16)
      XPPPreload(3, &p1[v+1], 16)
      XPPPreload(4, &p1[v+1], 16)
      XPPPreload(5, &p1[v+2], 16)
      XPPPreload(6, &p1[v+2], 16)
      XPPPreload(7, &p1[v+2], 16)
      XPPPreloadClean(3, &p2[v+1], 16)
      XPPExecute(config, IRAM(0), IRAM(1), IRAM(2), IRAM(3))
                         IRAM(4), IRAM(5), IRAM(6), IRAM(7))

}
```

for data duplication, respectively.

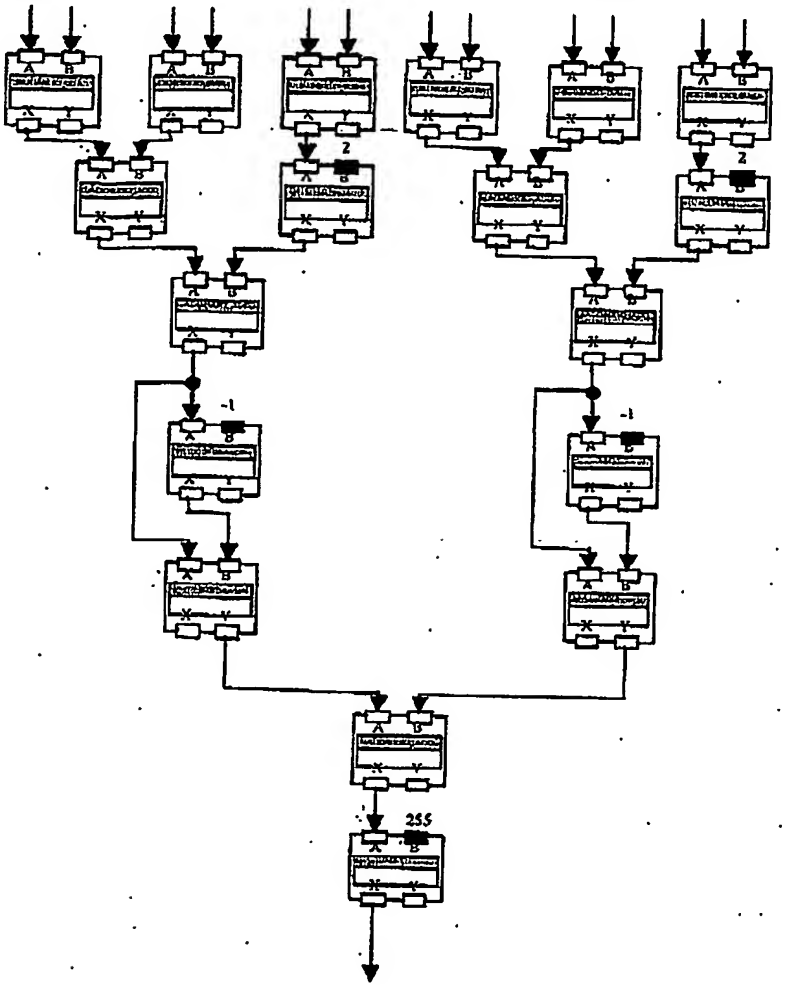


Figure 57 The main calculation network of the edge3x3 configuration. The MULT-SORT combination does the abs() calculation while the SORT does the min() calculation.

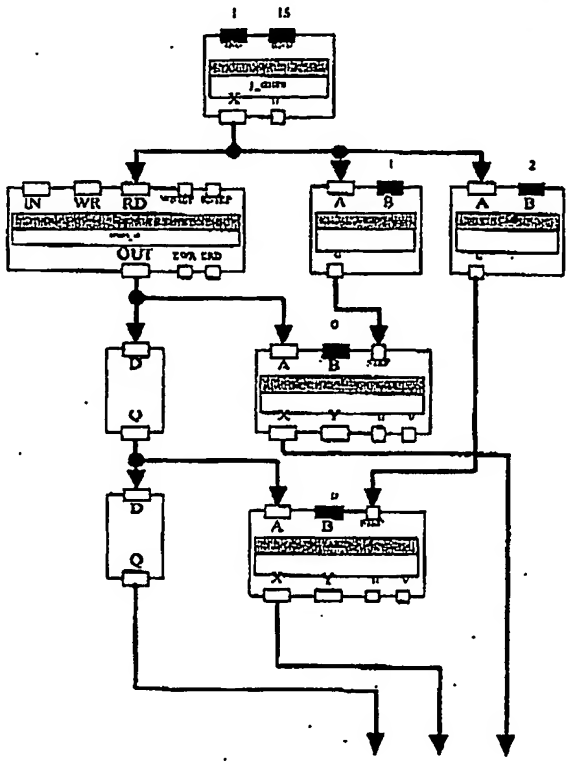


Figure 58 One input after shift register synthesis. The leftmost input contains p1[][h], the middle one p1[][h+1] and the rightmost p1[][h+2], respectively.

The values for place & route and simulation are compared in the following table. Note that a common RISC DSP with two MAC units and hardware loop support needs about 4000 cycles for the same code. This comparison does not account for cache misses. Furthermore it is obvious, that the number of input values is very small in this example and the DSP calculation time is proportional to that number. The XPP performance on the other hand will improve with the number of input values. Therefore the XPP performance will be more impressive with bigger image sizes.

Parameter	Value (shift register synthesis)			Value (data duplication)		
Vector length	16			16		
Reused data set size	256			256		
I/O IRAMs	3 I + 1 O = 4			8 I + 1 O = 9		
ALU	27			21		
BREG	21 (1 defined + 20 route)			10 (1 defined + 9 route)		
FREG	22 (9 defined + 23 route)			19 (3 defined + 16 route)		
Data flow graph width	14			14		
Data flow graph height	3 (shift registers) + 8 (calculation)			8 (calculation)		
Configuration cycles (simulated)	configuration		2262	configuration		2145
	preloads[1]	14*3*4	168	preloads	8*8*4	256
	cycles	14*57	798	cycles	14*52	728
	sum		3228	sum		3129

[1] assuming 4 words/cycle burst transfer

5.1.5 Enhancing Parallelism

After the synthesis the configuration calculating the inner loop utilizes 27 ALUs and 4 IRAMs for shift register synthesis and 21 ALUs and 9 IRAMs for data duplication, respectively. Assuming a XPP64 core this leaves plenty of room for further optimizations. Nevertheless, since all optimizations enhancing parallelism are performed before the synthesis takes place, it is crucial that they estimate the needed resources and the benefit of the transformation very carefully. Furthermore they have to account for both input preparation strategies to estimate correct values.

Loop Unrolling

Fully unrolling the inner loop would not lead to satisfying results, because the number of inputs and outputs increases dramatically. That means data duplication would not be applicable and shift register synthesis would exhaust most of the benefits of the parallelism by producing a very long pipeline for each data flow graph. Although partial unrolling of the inner loop would be applicable it promises not much benefit for the area penalty introduced.

Loop unrolling the outer loop is also not applicable since it produces a further configuration. Nevertheless a related transformation could do a good job on this loop nest.

Unroll-and-Jam

The unroll-and-jam algorithm enhances parallelism and also improves IRAM usage. It brings pairs of iterations together ideally reusing IRAM outputs and calculation results. The algorithm partially unrolls the outer loop and fuses the originated inner loops. Before the unroll-and-jam is performed the so-called unroll-and-jam factor must be determined which denominates the unrolling factor of the outer loop. This is mainly influenced by the number of ALUs n (= 64 assuming XPP64) and calculates to $c_{\text{unroll-and-jam}} = \frac{n_{XPP}}{n_{\text{inner loop}}} = \frac{64}{27} = 2$ (integer division).

Thus the source code would be transformed to.

```
for(v=0; v<=VERLEN-3; v+=2) {
    for(h=0; h<=HORLEN-3; h++) {
        p2[v+1][h+1] = min( abs((p1[v+2][h] - p1[v][h]) +
                                (p1[v+2][h+2] - p1[v][h+2]) +
                            2 * (p1[v+2][h+1] - p1[v][h+1])) +
                            abs((p1[v][h+2] - p1[v][h]) +
                                (p1[v+2][h+2] - p1[v+2][h]) +
                            2 * (p1[v+1][h+2] - p1[v+1][h])), 255);
        p2[v+2][h+1] = min( abs((p1[v+3][h] - p1[v+1][h]) +
                                (p1[v+3][h+2] - p1[v+1][h+2]) +
                            2 * (p1[v+3][h+1] - p1[v+1][h+1])) +
                            abs((p1[v+1][h+2] - p1[v+1][h]) +
                                (p1[v+3][h+2] - p1[v+3][h]) +
                            2 * (p1[v+2][h+2] - p1[v+2][h])), 255);
    }
}
```

The transformation introduces additional accesses to p1[v+3][h], p1[v+3][h+2], p1[v+3][h+1], and p1[v+1][h+1] (the former hole in the access pattern) as well as a write access to p2[v+2][h+1]. That means 2 IRAMs more for shift register synthesis (one input, one output) and 5 IRAMs more for data duplication (4 input, 1 output), while performance is doubled.

Parameter	Value (shift register synthesis)			Value (data duplication - no IRAM placement)		
Vector length	16			16		
Reused data set size	256			256		
I/O IRAMs	4 I + 2 O = 6			12 I + 2 O = 14		
ALU	45			37		
BREG	31 (12 defined + 19 route)			42 (4 defined + 38 route)		
FREG	29 (1 defined + 28 route)			18 (1 defined + 17 route)		
Data flow graph width	14			14		
Data flow graph height	3 (shift registers) + 8 (calculation)			8 (calculation)		
Configuration cycles (simulated)	configuration		2753	configuration		2754
	preloads	7*4*4	112	preloads	7*12*4	336
	cycles	7*53	371	cycles	7*69	483
	sum		3236	sum		3573

Parameter	Value (data duplication - with IRAM placement)				
Vector length	16				
Reused data set size	256				
I/O IRAMs	12 I + 2 O = 14				
ALU	37				
BREG	36 (4 defined + 32 route)				
FREG	24 (1 defined + 23 route)				
Data flow graph width	14				
Data flow graph height	3 (shift registers) + 8 (calculation)				
Configuration cycles (simulated)	configuration		2768		
	preloads	7*12*4	336		
	cycles	7*51	357		
	sum		3461		

The simulated results are shown in the table above. Please note the differences of the two columns labeled with "data duplication". The first used xmap to place the IRAMs, while in the second the IRAMs were placed by hand using a greedy algorithm which places IRAMs that are operands of the same operator in one line (as long as this is possible). The second solution improved the iteration cycles by 18. This shows that IRAM placement has a great impact to the final performance.

The traditional unroll-and-jam algorithm uses loop peeling to split the outer loop in a preloop and an unroll-able main loop to handle odd loop counts. When we assume for instance $n = 128$ the unroll-and-jam factor would calculate to

$$c_{\text{unroll-and-jam}} = \frac{128}{27} = 4.$$

Since the outer loop count (14) is not a multiple of 4, the algorithm virtually peels off the first two iterations and fuses the two loops at the end adding guards to the inner loop body. Then the code looks like (guards *emphasized*)

```
  for(v=0; v<=VERLEN-5; v+=4) {
       for(h=0; h<=HORLEN-3; h++) {
            p2[v+1][h+1] = min( abs((p1[v+2][h] - p1[v][h]) +
                                    (p1[v+2][h+2] - p1[v][h+2]) +
                                2 * (p1[v+2][h+1] - p1[v][h+1])) +
                                abs((p1[v][h+2] - p1[v][h]) +
                                    (p1[v+2][h+2] - p1[v+2][h])) +
                                2 * (p1[v+1][h+2] - p1[v+1][h])), 255);
            p2[v+2][h+1] = min( abs((p1[v+3][h] - p1[v+1][h]) +
                                    (p1[v+3][h+2] - p1[v+1][h+2]) +
                                2 * (p1[v+3][h+1] - p1[v+1][h+1])) +
                                abs((p1[v+1][h+2] - p1[v+1][h]) +
                                    (p1[v+3][h+2] - p1[v+3][h]) +
                                2 * (p1[v+2][h+2] - p1[v+2][h])), 255);
if(v>0)     p2[v+3][h+1] = min( abs((p1[v+4][h] - p1[v+2][h]) +
                                    (p1[v+4][h+2] - p1[v+2][h+2]) +
                                2 * (p1[v+4][h+1] - p1[v+2][h+1])) +
                                abs((p1[v+2][h+2] - p1[v+2][h]) +
                                    (p1[v+4][h+2] - p1[v+4][h]) +
                                2 * (p1[v+3][h+2] - p1[v+3][h])), 255);
if(v>1)     p2[v+4][h+1] = min( abs((p1[v+5][h] - p1[v+3][h]) +
                                    (p1[v+5][h+2] - p1[v+3][h+2]) +
                                2 * (p1[v+5][h+1] - p1[v+3][h+1])) +
                                abs((p1[v+3][h+2] - p1[v+3][h]) +
                                    (p1[v+5][h+2] - p1[v+5][h]) +
                                2 * (p1[v+4][h+2] - p1[v+4][h])), 255);
       }
  }
```

5.1.6 Parameterized Function

Source code

The benchmark source code is not very likely to be written in that form in real world applications. Normally it would be encapsulated in a function with parameters for input and output arrays along with the sizes of the picture to work on.

Therefore the source code would look similar to:

```
void edge3x3(int *p1, int *p2, int HORLEN, int VERLEN)
{
  for(v=0; v<=VERLEN-3; v++) {
    for(h=0; h<=HORLEN-3; h++) {
      htmp = (**(p1 + (v+2) * HORLEN + h)   - **(p1 + v * HORLEN + h)) +
             (**(p1 + (v+2) * HORLEN + h+2) - **(p1 + v * HORLEN + h+2))+
        2 * (**(p1 + (v+2) * HORLEN + h+1) - **(p1 + v * HORLEN + h+1));
      if (htmp < 0)
          htmp = - htmp;
      vtmp = (**(p1 +     v * HORLEN + h+2) - **(p1 + v * HORLEN + h)) +
             (**(p1 + (v+2) * HORLEN + h+2) - **(p1 + (v+2) * HORLEN + h))+
        2 * (**(p1 + (v+1) * HORLEN + h+2) - **(p1 + (v+1) * HORLEN + h));
      if (vtmp < 0)
          vtmp = - vtmp;

      sum = htmp + vtmp;
```

```
        if (sum > 255)
              sum = 255;
        **(p2 + (v+1) * HORLEN + h+1) = sum;
      }
    }}
```

This requires some additional features from the compiler.

- interprocedural optimizations and analysis
- hints by the Programmer (e.g. a compiler known assert(VERLEN % 2 == 0) makes unroll-and-jam actually possible without peeling off iterations and running them conditionally)

Fitting the Algorithm Optimally to the Array

Since HORLEN and VERLEN are not known at compile time these unknown parameters introduce some constraints which prevent pipeline vectorization. The compiler must assume that the IRAMs cannot hold all HORLEN input values in a row, so pipeline vectorization would not be possible.

Strip Mining Inner Loop

Strip mining partitions the inner loop into a loop that runs over a strip, which is chosen to be of the same size as the IRAMs can hold and a by strip loop iterating over the strips. Of course the strip loops upper bound must be adjusted for the possible incomplete last strip. After the strip mining the original code would look like (outer v-loop neglected):

```
for(h=0; h <= HORLEN-3; h+= stripsize)
  for(hh=h; h<=min(h+stripsize-1, HORLEN-3); hh++) {
    htmp = (**(p1 + (v+2) * HORLEN + hh)   - **(p1 + v * HORLEN + hh)) +
    ............
  }
}
```

Assuming a IRAM size strip size of 256 the following simulated results can be obtained for one strip. The values must be multiplied with the number of strips to be calculated.

Parameter	Value (shift register synthesis)			Value (data duplication - with IRAM placement)		
Vector length	16			16		
Reused data set size	256			256		
I/O IRAMs	4 I + 2 O = 6			12 I + 2 O = 14		
ALU	45			37		
BREG	31 (12 defined + 19 route)			42 (4 defined + 38 route)		
FREG	29 (1 defined + 28 route)			18 (1 defined + 17 route)		
Data flow graph width	14			14		
Data flow graph height	3 (shift registers) + 8 (calculation)			8 (calculation)		
Configuration cycles (simulated)	configuration		2753	configuration		2754
	preloads	7*4*64	1792	preloads	7*12*64	5376
	cycles	128*530	67840	cycles	128*553	70784
	sum		72385	sum		78914

The RISC DSP needs about 1.47 million cycles for this amount of data. As mentioned above these values do not include cache miss penalties and truly underestimate the real values. Furthermore it can be seen that data duplication does not improve the performance. The reason for this seems to be an worse placement and routing.

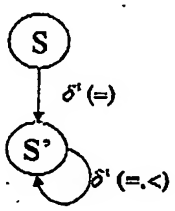
5.2 FIR Filter

5.2.1 Original Code

Source code:

```
#define N 256
#define M 8

for (i = 0; i < N-M+1; i++) {
S:  y[i] = 0;
  for (j = 0; j < M; j++)
S':   y[i] += c[j] * x[i+M-j-1];
}
```

The constants N and M are replaced by their values by the pre-processor. The data dependence graph is the following:




```
for (i = 0; i < 269; i++) {
  S:  y[i] = 0;
    for (j = 0; j < 8; j++)
  S':   y[i] += c[j] * x[i+7-j];
  }
```

We have the following table:

Parameter	Value
Vector length	269
Reused data set size	-
I/O IRAMs	3
ALU	2
BREG	0
FREG	0
Data flow graph width	1
Data flow graph height	2
Configuration cycles	2+8=10

5.2.2 First Solution

In the case we want to save memory, the straightforward solution is to unroll the inner loop and to use shift register synthesis to delay the values of array x in the pipeline. No other optimization is applied before as either they do not have an effect on the loop or they increase the need for IRAMs. After loop unrolling, we obtain the following code:

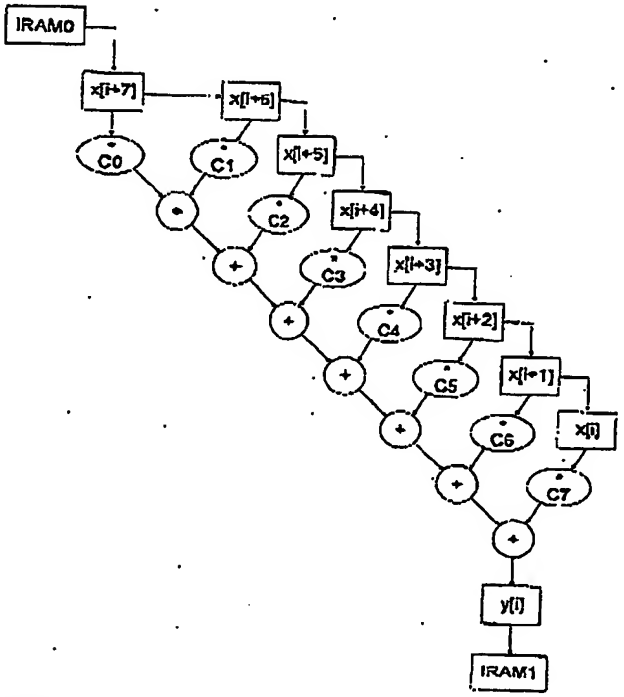
```
for (i = 0; i < 269; i++) {
    y[i] = 0;
    y[i] += c[0] * x[i+7];
    y[i] += c[1] * x[i+6];
    y[i] += c[2] * x[i+5];
    y[i] += c[3] * x[i+4];
    y[i] += c[4] * x[i+3];
    y[i] += c[5] * x[i+2];
    y[i] += c[6] * x[i+1];
    y[i] += c[7] * x[i];
}
```

Then the table looks like this:

Parameter	Value
Vector length	269
Reused data set size	-
I/O IRAMs	9
ALU	16
BREG	0
FREG	0
Data flow graph width	2
Data flow graph height	9
Configuration cycles	9+269=278

Dataflow analysis reveals that *y[0]=f(x[0],...,x[7]), y[1]=f(x[1],...,x[8]),...,y[i]=f(x[i],...,x[i+7])*. Successive values of *y* depend on almost the same successive values of *x*. To prevent unnecessary accesses to the IRAMs, the values of *x* needed for the computation of the next values of *y* are kept in registers. In our case this shift register synthesis needs 7 registers. This will be achieved on the PACT XPP, by keeping them into FREGs. Then we obtain the dataflow graph depicted below. An IRAM is used for the input values and an IRAM for the output values. The first 8 cycles are used to fill the pipeline and then the throughput is of one output value/cycle. We can depict the code as the following:

```
r0 = x[0];
r1 = x[1];
r2 = x[2];
r3 = x[3];
r4 = x[4];
r5 = x[5];
r6 = x[6];
r7 = x[7];
for (i = 0; i < 269; i++) {
  y[i] = c7*r0 + c6*r1 + c5*r2 + c4*r3 + c3*r4 + c2*r5 + c1*r6 + c0*r7;
  r0 = r1;
  r1 = r2;
  r2 = r3;
  r3 = r4;
  r4 = r5;
  r5 = r6;
  r6 = r7;
  r7 = x[i+7];
  }
```



The final table is shown below, and the expected speedup with respect to a standard superscalar processor with 2 instructions issued per cycle is 13.6.

Parameter	Value
Vector length	269
Reused data set size	-
I/O IRAMs	2
ALU	16
BREG	0
FREG	7
Data flow graph width	3
Data flow graph height	9
Configuration cycles	8+269=277

Ops	Number
LD/ST (2 cycles)	2
ADDRCOMP (1 cycle)	0
ADD/SUB (1 cycle)	8
MUL (2 cycles)	8
SHIFT (1 cycle)	0
Cycles per iteration	28
Cycles needed for the loop (2-way)	(28*269)/2=3766

Variant with Larger Loop Bounds

Let us take larger loop bounds and set the values of N and M to 1024 and 64.

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (j = 0; j < 64; j++)
      y[i] += c[j] * x[i+63-j];
  }
```

Following the loop optimizations driver given before, we apply loop tiling to reduce the iteration range of the inner loop. We obtain the following loop nest.

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (jj = 0; jj < 8; jj++)
      for (j = 0;j < 8;j++)
          y[i] += c[8*jj+j] * x[i+63-8*jj-j];
}
```

A subsequent application of loop unrolling on the inner loop yields:

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (jj = 0; jj < 8; jj++) {
        y[i] += c[8*jj] * x[i+63-8*jj];
        y[i] += c[8*jj+1] * x[i+62-8*jj];
```

```
        y[i] += c[8*jj+2] * x[i+61-8*jj];
        y[i] += c[8*jj+3] * x[i+60-8*jj];
        y[i] += c[8*jj+4] * x[i+59-8*jj];
        y[i] += c[8*jj+5] * x[i+58-8*jj];
        y[i] += c[8*jj+6] * x[i+57-8*jj];
        y[i] += c[8*jj+7] * x[i+56-8*jj];
    }
```

Finally we obtain the same dataflow graph as above, except that the coefficients must be read from another IRAM rather than being directly handled like constants by the multiplications. After shift register synthesis the code is the following:

```
for (i = 0; i < 961; i++) {
  r0 = x[i+56];
  r1 = x[i+57];
  r2 = x[i+58];
  r3 = x[i+59];
  r4 = x[i+60];
  r5 = x[i+61];
  r6 = x[i+62];
  r7 = x[i+63];
  for (jj = 0; jj < 8; jj++)
     y[i] = c[8*jj]*r0 + c[8*jj+1]*r1 + c[8*jj+2]*r2 + c[8*jj+3]*r3 +
            c[8*jj+4]*r4 + c[8*jj+5]*r5 + c[8*jj+6]*r6 + c[8*jj+7]*r7;
     r0 = r1;
     r1 = r2;
     r2 = r3;
     r3 = r4;
     r4 = r5;
     r5 = r6;
     r6 = r7;
     r7 = x[i+63-8*jj];
  }
}
```

The table is the same than before except for the vector length and the expected speedup with respect to a standard superscalar processor with 2 instructions issued per cycle is 17.5.

Parameter	Value
Vector length	8
Reused data set size	-
I/O IRAMs	2
ALU	16
BREG	0
FREG	7
Data flow graph width	3
Data flow graph height	9
Configuration cycles	8+8=16

Ops	Number
LD/ST (2 cycles)	10
ADDRCOMP (1 cycle)	0
ADD/SUB (1 cycle)	16
MUL (2 cycles)	17
SHIFT (1 cycle)	0
Cycles per iteration	70
Cycles needed for the loop (2-way)	(70*8)/2=280

5.2.3 A More Parallel Solution

The solution we presented does not expose a lot of parallelism in the loop. We can try to explicitly parallelize the loop before we generate the dataflow graph. Of course exposing more parallelism means more pressure on the memory hierarchy.

In the data dependence graph presented at the beginning, the only loop-carried dependence is the dependence on *S'* and it is only caused by the reference to *y[i]*. Hence we apply node splitting to get a more suitable data dependence graph. We obtain then:

```
for (i = 0; i < 249; i++) {
    y[i] = 0;
    for (j = 0; j < 8; j++)
      {
        tmp = c[j] * x[i+7-j];
        y[i] += tmp;
      }
}
```

Then scalar expansion is performed on *tmp* to remove the anti loop-carried dependence caused by it, and we have the following code:

```
for (i = 0; i < 249; i++) {
    y[i] = 0;
    for (j = 0; j < 8; j++)
      {
        tmp[j] = c[j] * x[i+7-j];
        y[i] += tmp[j];
      }
}
```

The parameter table is the following:

Parameter	Value
Vector length	249
Reused data set size	-
I/O IRAMs	3
ALU	2
BREG	0
FREG	1
Data flow graph width	2
Data flow graph height	2
Configuration cycles	2+8=10

Then we apply loop distribution to get a vectorizable and a not vectorizable loop.

```
for (i = 0; i < 249; i++) {
    y[i] = 0;
    for (j = 0; j < 8; j++)
      tmp[j] = c[j] * x[i+7-j];
    for (j = 0; j < 8; j++)
        y[i] += tmp[j];
    }
}
```

The parameter table given below corresponds to the two inner loops in order to be compared with the preceding table.

Parameter	Value
Vector length	249
Reused data set size	-
I/O IRAMs	5
ALU	2
BREG	0
FREG	1
Data flow graph width	1
Data flow graph height	3
Configuration cycles	1*8+1*8=16

Then we must take into account the architecture. The first loop is fully parallel; this means that we would need 2*8=16 input values at a time. This is all right, as it corresponds to the number of IRAMS of the PACT XPP. Hence we do not need to strip-mine the first inner loop. The case of the second loop is trivial, it does not need to be strip-mined either. The second loop is a reduction, it computes the sum of a vector. This is easily found by the reduction recognition optimization and we obtain the following code.

```
for (i = 0; i < 249; i++) {
    y[i] = 0;
    for (j = 0; j < 8; j++)
       tmp[j] = c[j] * x[i+7-j];

    /* load the partial sums from memory using a shorter vector length */
    for (j = 0; j < 4; j++)
       aux[j] = tmp[2*j] + tmp[2*j+1];

    /* accumulate the short vector */
    for (j = 0;j < 1; j++)
        aux[2*j] = aux[2*j] + aux[2*j+1];

    /* sequence of scalar instructions to add up the partial sums */
    y[i] = aux[0] + aux[2];
}
```

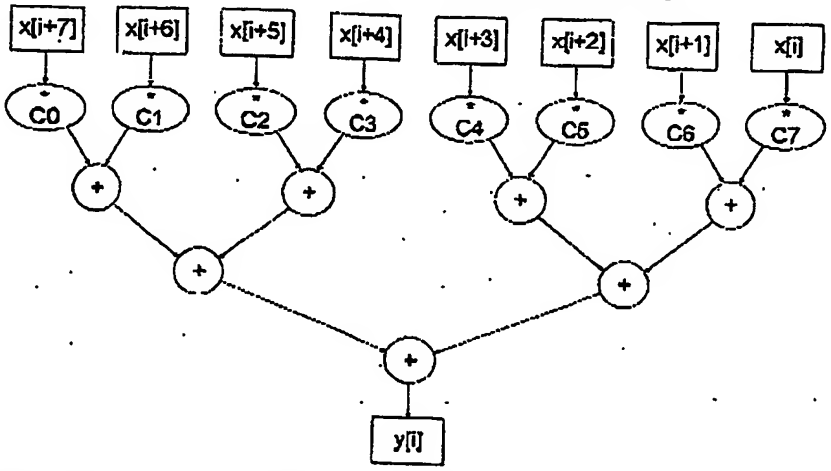
Like above we give only one table for all innermost loops and the last instruction computing *y[i]*.

Parameter	Value
Vector length	249
Reused data set size	-
I/O IRAMs	12
ALU	4
BREG	0
FREG	0
Data flow graph width	1
Data flow graph height	4
Configuration cycles	1*8+1*4+1*1=13

Finally loop unrolling is applied on the inner loops, the number of operations is always less than the number of processing elements of the PACT XPP.

```
for (i = 0; i < 961; i++)
   {
    tmp[0] = c[0] * x[i+7];
    tmp[1] = c[1] * x[i+6];
    tmp[2] = c[2] * x[i+5];
    tmp[3] = c[3] * x[i+4];
    tmp[4] = c[4] * x[i+3];
    tmp[5] = c[5] * x[i+2];
    tmp[6] = c[6] * x[i+1];
    tmp[7] = c[7] * x[i];

    aux[0] = tmp[0] + tmp[1];
    aux[1] = tmp[2] + tmp[3];
    aux[2] = tmp[4] + tmp[5];
    aux[3] = tmp[6] + tmp[7];

    aux[0] = aux[0] + aux[1];
    aux[2] = aux[2] + aux[3];

    y[i] = aux[0] + aux[2];
}
```

We obtain then the following dataflow graph representing the inner loop.



It could be mapped on the PACT XPP with each layer executed in parallel, thus needing 4 cycles/iteration and 15 ALU-PAEs, 8 of which needed in parallel. As the graph is already synchronized, the throughput reaches one iteration/cycle, after 4 cycles to fill the pipeline. The coefficients are taken as constant inputs by the ALUs performing the multiplications.

The drawback of this solution is that it uses 16 IRAMs, and that the input data must be stored in a special order. The number of needed IRAMs can be reduced if the coefficients are handled like constant for each ALU. But due to data locality of the program, we can assume that the data already reside in the cache. And as the transfer of data from the cache to the IRAMs can be achieved efficiently, the configuration can be executed on the PACT XPP without waiting for the data to be ready in the IRAMs. The parameter table is then the following:

Parameter	Value
Vector length	249
Reused data set size	-
I/O IRAMs	16
ALU	15
BREG	0
FREG	0
Data flow graph width	8
Data flow graph height	4
Configuration cycles	4+961

Variant with Larger Bounds

To make the things a bit more interesting, we set the values of N and M to 1024 and 64.

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (j = 0; j < 64; j++)
      y[i] += c[j] * x[i+63-j];
}
```

The data dependence graph is the same as above. We apply then node splitting to get a more convenient data dependence graph.

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (j = 0; j < 64; j++)
      {
        tmp = c[j] * x[i+63-j];
        y[i] += tmp;
      }
}
```

After scalar expansion:

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (j = 0; j < 64; j++)
      {
        tmp[j] = c[j] * x[i+63-j];
        y[i] += tmp[j];
      }
  }
```

After loop distribution:

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (j = 0; j < 64; j++)
      tmp[j] = c[j] * x[i+63-j];
    for (j = 0; j < 64; j++)
        y[i] += tmp[j];
      }
  }
```

We go through the compiling process, and we arrive to the set of optimizations that depends upon architectural parameters. We want to split the iteration space, as too many operations would have to be performed in parallel, if we keep it as such. Hence we perform strip-mining on the 2 loops. We can only access 16 data at a time, so, because of the first loop, the factor will be $64 * 2/16 = 8$ for the 2 loops (as we always have in mind that we want to execute both at the same time on the PACT XPP).

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (jj = 0; jj < 8; jj++)
      for (j=0;j < 8; j++)
        tmp[8*jj+j] = c[8*jj+j] * x[i+63-8*jj-j];
    for (jj = 0; jj < 8  ; jj++)
      for (j=0;j < 8; j++)
          y[i] += tmp[8*jj+j];
}
```

And then loop fusion on the *jj* loops is performed.

```
for (i = 0; i < 961; i++) {
    y[i] = 0;
    for (jj = 0; jj < 8; jj++) {
      for (j=0;j < 8;j++)
          tmp[8*jj+j] = c[8*jj+j] * x[i+63-8*jj-j];
      for (j=0;j < 8;j++)
          y[i] += tmp[8*jj+j];
    }
}
```

Now we apply reduction recognition on the second innermost loop.

```
for (i = 0; i < 961; i++) {
    tmp = 0;
    for (jj = 0; jj < 8; jj++)
      {
      for (j = 0; j < 8; j++)
         tmp[8*jj+j] = c[8*jj+j] * x[i+63-8*jj-j];

      /* load the partial sums from memory using a shorter vector length */
        for (j = 0; j < 4; j++)
          aux[j] = tmp[8*jj+2*j] + tmp[8*jj+2*j+1];

      /* accumulate the short vector */
        for (j = 0;j < 1; j++)
          aux[2*j] = aux[2*j] + aux[2*j+1];

      /* sequence of scalar instructions to add up the partial sums */
        y[i] = aux[0] + aux[2];

}
```

And then loop unrolling.

```
for (i = 0; i < 961; i++)
    for (jj = 0; jj < 8; jj++)
      {
        tmp[8*jj] = c[8*jj]   * x[i+63-8*jj];
        tmp[8*jj+1] = c[8*jj+1] * x[i+62-8*jj];
        tmp[8*jj+2] = c[8*jj+2] * x[i+61-8*jj];
        tmp[8*jj+3] = c[8*jj+3] * x[i+59-8*jj];
        tmp[8*jj+4] = c[8*jj+4] * x[i+58-8*jj];
        tmp[8*jj+5] = c[8*jj+5] * x[i+57-8*jj];
        tmp[8*jj+6] = c[8*jj+6] * x[i+56-8*jj];
        tmp[8*jj+7] = c[8*jj+7] * x[i+55-8*jj];

        aux[0] = tmp[8*jj] + tmp[8*jj+1];
        aux[1] = tmp[8*jj+2] + tmp[8*jj+3];
        aux[2] = tmp[8*jj+4] + tmp[8*jj+5];
        aux[3] = tmp[8*jj+6] + tmp[8*jj+7];

        aux[0] = aux[0] + aux[1];
        aux[2] = aux[2] + aux[3];

        y[i] = aux[0] + aux[2];
    }
```

We implement the innermost loop on the PACT XPP directly with a counter. The IRAMs are used in FIFO mode, and filled according to the addresses of the arrays in the loop. IRAM0, IRAM2, IRAM4, IRAM6 and IRAM8 contain array c. IRAM1, IRAM3, IRAM5 and IRAM7 contain array *x*. Array *c* contains 64 elements, that is each IRAM contains 8 elements. Array *x* contains 1024 elements, that is 128 elements for each IRAM. Array *y* is directly written to memory, as it is a global array and its address is constant. This constant is used to initialize the address counter of the configuration. The final parameter table is the following:

Parameter	Value
Vector length	8
Reused data set size	-
I/O IRAMs	16
ALU	15
BREG	0
FREG	0
Data flow graph width	8
Data flow graph height	4
Configuration cycles	4+8=12

Nevertheless it should be noted that this version should be less efficient than the previous one. As the same data must be loaded in the different IRAMs from the cache, we have a lot of transfers to achieve before the configuration can begin the computations. This overhead must be taken into account by the compiler when choosing the code generation strategy. This means also that the first solution is the solution that will be chosen by the compiler.

5.2.4 Other Variant

Source Code

```
for (i = 0; i < N-M+1; i++) {
    tmp = 0;
    for (j = 0; j < M; j++)
      tmp += c[j] * x[i+M-j-1];
    x[i] = tmp;
  }
```

In this case, it is trivial that the data dependence graph is cyclic due to dependences on *tmp*. Therefore scalar expansion is applied on the loop, and we obtain in fact the same code as the first version of the FIR filter as shown below.

```
for (i = 0; i < N-M+1; i++) {
    tmp[i] = 0;
    for (j = 0; j < M; j++)
      tmp[i] += c[j] * x[i+M-j-1];
    x[i] = tmp[i];
  }
```

5.3 Matrix Multiplication

5.3.1 Original Code

Source code:

```
#define L 10
#define M 15
#define N 20

int A[L][M];
int B[M][N];
int R[L][N];

main() {
  int i, j, k, tmp, aux;

  /* input A (L*M values) */
  for(i=0; i<L; i++)
      for(j=0; j<M; j++)
          scanf("%d", &A[i][j]);

  /* input B (M*N values) */
  for(i=0; i<M; i++)
      for(j=0; j<N; j++)
          scanf("%d", &B[i][j]);

  /* multiply */
  for(i=0; i<L;i++)
     for(j=0; j<N; j++) {
          aux = 0;
          for(k=0; k<M; k++)
              aux += A[i][k] * B[k][j];
          R[i][j] = aux;
      }

  /* write data stream */
  for(i=0; i<L; i++)
      for(j=0; j<N; j++)
          printf("%d\n", R[i][j]);
}
```

5.3.2 Preliminary Transformations

Since no inline-able function calls are present, no interprocedural code movement is done.

Of the four loop nests the one with the "/* multiply */" comment is the only candidate for running partly on the XPP. All others have function calls in the loop body and are therefore discarded as candidates very early in the compiler.

Dependency Analysis

```
    for(i=0; i<L;i++)
        for(j=0; j<N; j++) {
S1          aux = 0;
            for(k=0; k<M; k++)
```

```
S2                    aux += A[i][k] * B[k][j];
S3              R[i][j] = aux;
        }
```

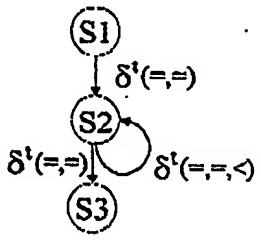


Figure 59 Data dependency graph for matrix multiplication

The data dependency graph shows no dependencies that prevent pipeline vectorization. The loop carried true dependence from S2 to itself can be handled by a feedback of aux as described in [1].

Reverse Loop-Invariant Code Motion

To get a perfect loop nest we move S1 and S3 inside the k-loop. Therefore appropriate guards are generated to protect the assignments. The code after this transformation looks like

```
for(i=0; i<L;i++)
      for(j=0; j<N; j++)
            for(k=0; k<M; k++) {
                  if (k == 0) aux = 0;
                  aux += A[i][k] * B[k][j];
                  if (k == M-1) R[i][j] = aux;
            }
```

Scalar Expansion

Our goal is to interchange the loop nests to improve the array accesses to utilize the cache best. Unfortunately the guarded statements involving aux cause backward loop carried anti-dependences carried by the j loop. Scalar expansion will break these dependences, allowing loop interchange.

```
for(i=0; i<L;i++)
      for(j=0; j<N; j++)
            for(k=0; k<M; k++) {
                  if (k == 0) aux[j] = 0;
                  aux[j] += A[i][k] * B[k][j];
                  if (k == M-1) R[i][j] = aux[j];
            }
```

Loop Interchange for Cache Reuse

Visualizing the main loop shows the iteration spaces for the array accesses (Figure 60). Since C arrays are placed in row major order the cache lines are placed in the array rows. At first sight there seems no need for optimization because the algorithm requires at least one array access to stride over a column. Nevertheless this assumption misses the fact that the access rate is of interest, too. Closer examination shows that array R is accessed in every j iteration, while B is accessed every k-iteration, always

producing a cache miss[2]. This leaves a possibility for loop interchange to improve cache access as proposed by Kennedy and Allen in [7].

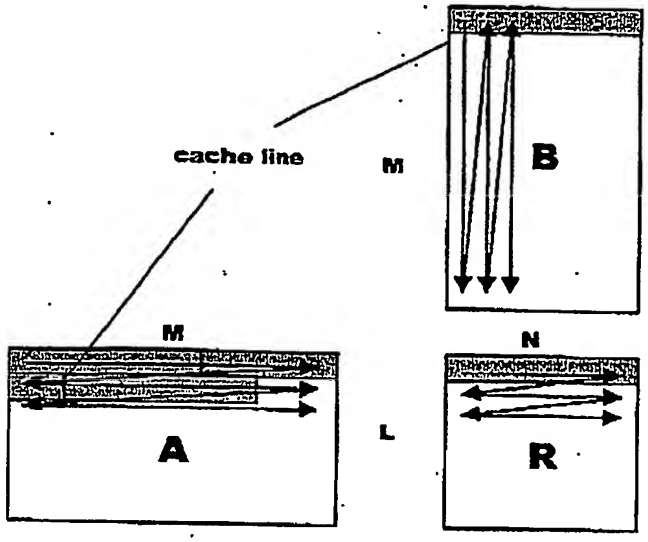


Figure 60 The visualized array access sequences.

Finding the best loop nest is relatively simple. The algorithm simply interchanges each loop of the nests into the innermost position and annotates it with the so-called innermost memory cost term. This cost term is a constant for known loop bounds or a function of the loop bound for unknown loop bounds. The term is calculated in three steps.

- First the cost of each reference[3] in the innermost loop body is calculated to
  - 1, if the reference does not depend on the loop induction variable of the (current) innermost loop
  - the loop count, if the reference depends on the loop induction variable and strides over a noncontiguous area in respect of the cache layout
  - $\frac{N \cdot s}{b}$, if the reference depends on the loop induction variable and strides over a contiguous dimension. In this case N is the loop count, s is the step size and b is the cache line size, respectively.
- Second each reference cost is weighted with a factor for each other loop, which is
  - 1, if the reference does not depend on the loop index
  - the loop count, if the reference depends on the loop index.
- Third the overall loop nest cost is calculated by summing the costs of all reference costs.

After invoking this algorithm for each loop as the innermost, the one with the lowest cost is chosen as the innermost, the next as the next outermost, and so on.

[2] We neglect "aux" in this observation since we do not expect it to be written to or read from memory (no defs or uses outside the loop nest)

[3] Reference means access to an array in this case. Since the transformation wants to optimize cache access, it must address references to the same array within small distances as one. This prohibits over-estimation of the actual costs.

Innermost loop	R[i][j]	A[i][k]	B[k][j]	Memory access cost
k	$1 \cdot L \cdot N$	$\frac{M}{b} \cdot L$	$M \cdot N$	$L \cdot N + \frac{M}{b} \cdot L + M \cdot N$
i	$1 \cdot L \cdot N$	$1 \cdot L \cdot M$	$1 \cdot M \cdot N$	$L \cdot N + L \cdot M + M \cdot N$
j	$\frac{N}{b} L$	$L \cdot M$	$\frac{N}{b} M$	$\frac{N}{b}(L+M) + L \cdot M$

Table 1 Loop memory access costs for the different loops being innermost

The table shows the values for the matrix multiplication. Since the j term is the smallest (of course assuming $b > 1$), the j-loop is chosen to be the innermost. The next outer loop then is k, and the outermost is i. Thus the resulting code after loop interchange is

```
for(i=0; i<L;i++)
      for(k=0; k<M; k++)
            for(j=0; j<N; j++) {
                  if (k == 0) aux[j] = 0;
                  aux[j] += A[i][k] * B[k][j];
                  if (k == M-1)R[i][j] = aux[j];
      }
```

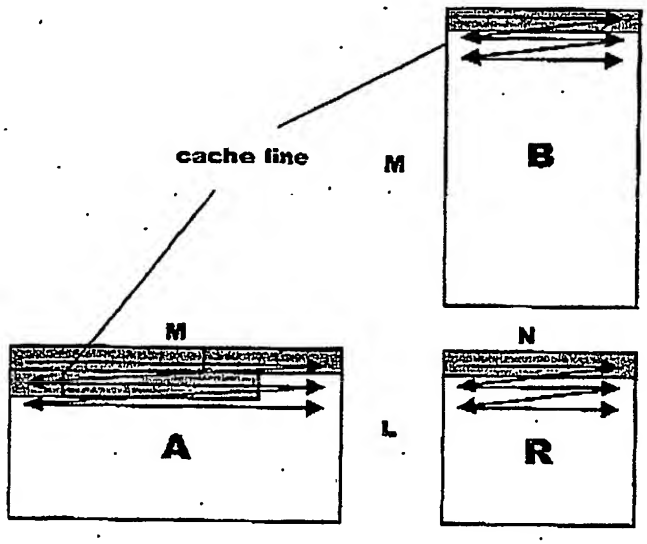


Figure 61 The visualized array access sequences after optimization. Here the improvement is visible to the naked eye, since array B is now read following the cache lines.

Figure 61 shows the improved iteration spaces. It is to say that this optimization does not optimize primarily for the XPP, but mainly optimizes the cache-hit rate, thus improving the overall performance.

Unroll and Jam

After improving the cache access behavior, the possibility for reduction recognition has been destroyed. This is a typical example for transformations where one excludes the other. Nevertheless we obtain more parallelism by doing unroll-and-jam. Therefore we unroll the outer loop partially with the unroll factor. This factor is mainly chosen by the minimum of two calculations.

- # available IRAMs / # used IRAMs in the inner loop body
- # available ALU resources / # used ALU resources in the inner loop

In this example the accesses to "A" and "B" depend on k (the loop which will be unrolled). Therefore they must be considered in the calculation. The accesses to "aux" and "R" do not depend on k. Thus they can be subtracted from the available IRAMs, but don not need to be added to the denominator. Therefore we calculate (assuming an XPP64) 14/2 = 7 for the unroll factor obtained by the IRAM resources.

On the other hand the loop body involves two ALU operations (1 add, 1 mult), which yields an unrolling factor of approximately 64/2 = 32[4]. The constraint generated by the IRAMs therefore dominates by far.

Having chosen the unroll factor we must trim our loop trip count to be a multiple of that factor. Since the k loop has a loop count of 15, we peel off the first iteration and unroll the remaining loop.

```
for(i=0; i<L;i++) {
    for(k=0; k<1; k++) {
        for(j=0; j<N; j++) {
            if (k==0) aux[j] = 0;
            aux[j] += A[i][k] * B[k][j];
            if (k==M-1) R[i][j] = aux[j];
        }
    }
    for(k=1; k<M; k+=7) {
        for(j=0; j<N; j++) {
            if (k==0) aux[j] = 0;
            aux[j] += A[i][k] * B[k][j];
            if (k==M-1) R[i][j] = aux[j];
        }
        for(j=0; j<N; j++) {
            if (k+1==0) aux[j] = 0;
            aux[j] += A[i][k+1] * B[k+1][j];
            if (k+1==M-1) R[i][j] = aux[j];
        }
        for(j=0; j<N; j++) {
            if (k+2==0) aux[j] = 0;
            aux[j] += A[i][k+2] * B[k+2][j];
            if (k+2==M-1) R[i][j] = aux[j];
        }
        for(j=0; j<N; j++) {
            if (k+3==0) aux[j] = 0;
            aux[j] += A[i][k+3] * B[k+3][j];
            if (k+3==M-1) R[i][j] = aux[j];
        }
        for(j=0; j<N; j++) {
            if (k+4==0) aux[j] = 0;
            aux[j] += A[i][k+4] * B[k+4][j];
            if (k+4==M-1) R[i][j] = aux[j];
```

[4] This is a very inaccurate estimation, since it neither estimates the resources spent by the controlling network, which decreases the unroll factor, nor takes it into account that e.g the BREG-PAEs also have an adder, which increases the unroll factor. Although it has no influence to this example the unroll factor calculation of course has to account for this in a production compiler.

```
            }
            for(j=0; j<N; j++) {
                  if (k+5==0) aux[j] = 0;
                  aux[j] += A[i][k+5] * B[k+5][j];
                  if (k+5==M-1) R[i][j] = aux[j];
            }
            for(j=0; j<N; j++) {
                  if (k+6==0) aux[j] = 0;
                  aux[j] += A[i][k+6] * B[k+6][j];
                  if (k+6==M-1) R[i][j] = aux[j];
            }
      }
}
```

Due to the fact that the reverse loop invariant code motion placed the loop invariant code into the inner loop which is now duplicated seven times, it is very likely that dead code elimination can get rid of some of these duplicates. Thus the code is shortened to

```
for(i=0; i<L;i++) {
      for(k=0; k<1; k++) {
            for(j=0; j<N; j++) {
                  if (k==0) aux[j] = 0;
                  aux[j] += A[i][k] * B[k][j];
            }
      }
      for(k=1; k<M; k+=7) {
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k] * B[k][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+1] * B[k+1][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+2] * B[k+2][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+3] * B[k+3][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+4] * B[k+4][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+5] * B[k+5][j];
            }
            for(j=0; j<N; j++) {
                  aux[j] += A[i][k+6] * B[k+6][j];
                  if (k+6==M-1) R[i][j] = aux[j];
            }
      }
}
```

Before we jam the inner loops we have to account for the fact that the first iteration of the k loop was peeled of which would produce an own configuration. Since we calculated the unroll-and-jam factor to fit into one configuration, this side effect has to be prevented. Because it should be no problem to run the k loop with variable step sizes, we fuse the k loops again and adjust the step size and guard the statements. This yields

```
for(i=0; i<L;i++) {
      for(k=0; k<M; k+= k<1 ? 1 : 7) {
            for(j=0; j<N; j++) {
                  if (k==0) aux[j] = 0;
                  if (k==0) aux[j] += A[i][k] * B[k][j];
            }
            for(j=0; j<N; j++) {
```

```
                if (k>0) aux[j] += A[i][k] * B[k][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+1] * B[k+1][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+2] * B[k+2][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+3] * B[k+3][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+4] * B[k+4][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+5] * B[k+5][j];
            }
            for(j=0; j<N; j++) {
                if (k>0) aux[j] += A[i][k+6] * B[k+6][j];
                if (k+6==M-1) R[i][j] = aux[j];
            }
        }
}
```

Now we can jam the inner loops and finally obtain

```
for(i=0; i<L;i++) {
      for(k=0; k<M; k+= k<1 ? 1 : 7) {
            for(j=0; j<N; j++) {
                if (k==0) aux[j] = 0;
                if (k==0) aux[j] += A[i][k] * B[k][j];
                if (k>0) {
                    aux[j] += A[i][k] * B[k][j];
                    aux[j] += A[i][k+1] * B[k+1][j];
                    aux[j] += A[i][k+2] * B[k+2][j];
                    aux[j] += A[i][k+3] * B[k+3][j];
                    aux[j] += A[i][k+4] * B[k+4][j];
                    aux[j] += A[i][k+5] * B[k+5][j];
                    aux[j] += A[i][k+6] * B[k+6][j];
                    if (k+6==M-1) R[i][j] = aux[j];
                }
            }
      }
}
```

5.3.3 XPP Code Generation

The innermost loop can be synthesized in a configuration, which uses 14 IRAMs for the input data, one IRAM to temporary store aux and one IRAM for the output array R. Furthermore it is necessary to pass the value of k to the XPP to direct the dataflow. This may be done by a streaming input. Figure 62 shows the dataflow graph of the synthesized configuration.

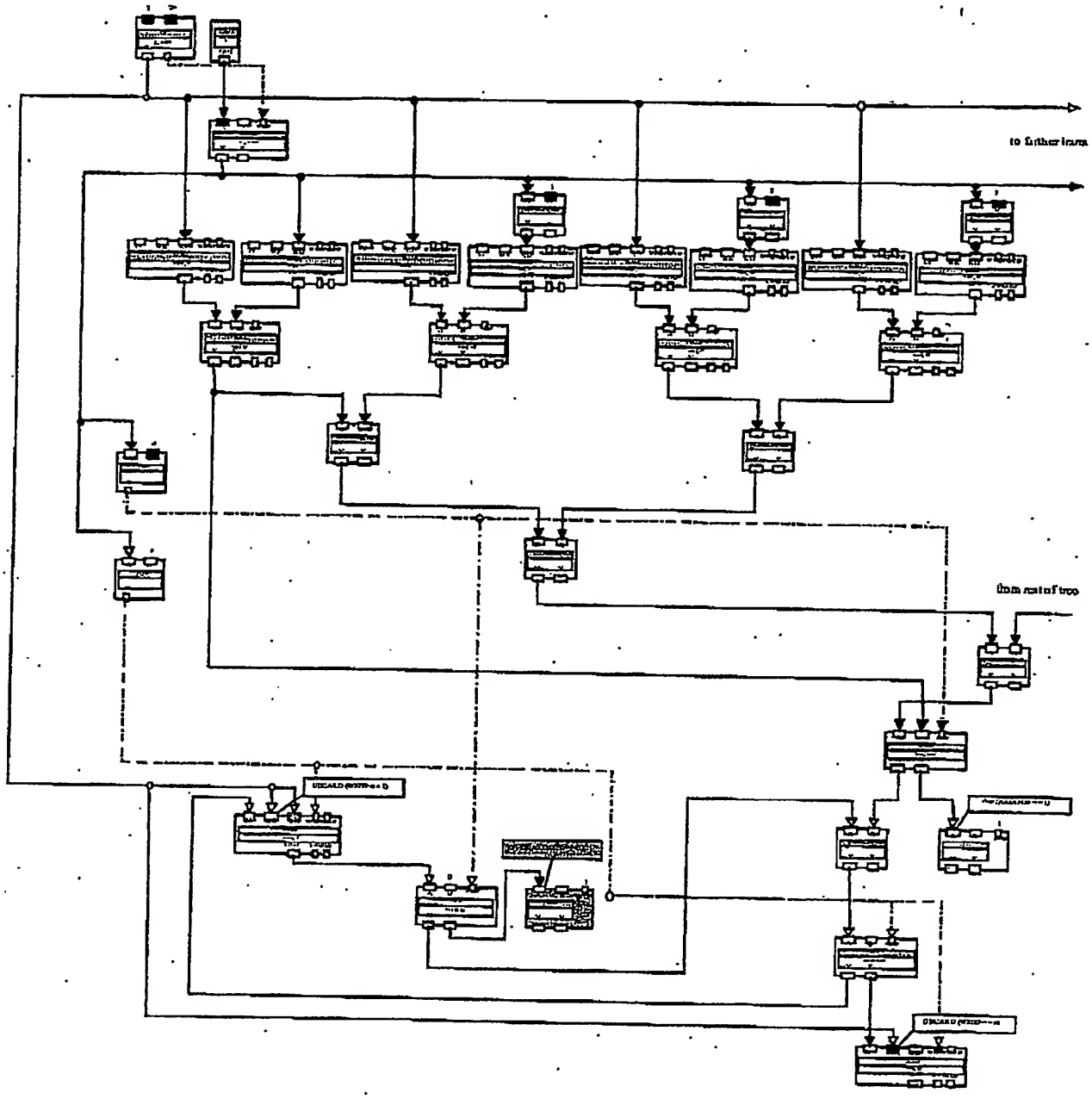


Figure 62 Dataflow graph of matrix multiplication after unroll and jam. The rightmost 3 branches are omitted. Event connections are emphasized in red color.

The following code shows the pseudo code executed on the RISC processor.

```
XPPPreload(config)
for(i=0; i<L;i++) {
      XPPPreload(0, &A[i][0], M)
      XPPPreload(1, &A[i][0], M)
      XPPPreload(2, &A[i][0], M)
      XPPPreload(3, &A[i][0], M)
      XPPPreload(4, &A[i][0], M)
      XPPPreload(5, &A[i][0], M)
      XPPPreload(6, &A[i][0], M)
      XPPPreloadClean(15, &R[i][0], M)
      for(k=0; k<M; k+= k<1 ? 1 : 7) {
            XPPPreload(7, &B[k][0], N)
            XPPPreload(8, &B[k+1][0], N)
            XPPPreload(9, &B[k+2][0], N)
            XPPPreload(10, &B[k+3][0], N)
            XPPPreload(11, &B[k+4][0], N)
            XPPPreload(12, &B[k+5][0], N)
            XPPPreload(13, &B[k+6][0], N)
            XPPExecute(config, IRAM(0), IRAM(1), IRAM(2), IRAM(3),
                               IRAM(4), IRAM(5), IRAM(6), IRAM(7),
                               IRAM(8), IRAM(9), IRAM(10), IRAM(11),
                               IRAM(12), IRAM(13), IRAM(15),k)
      }
}
```

The table shows the simulated configuration. The complete multiplication needs about 3120 cycles without the preloading and configuration. A typical RISC-DSP core with two MAC units and hardware loop support needs over 26000 cycles (when data is in zero-latency internal memory). Although the time for preloads and cache misses is neglected here, the values promise improvements of 200-300 percent compared to a standalone RISC core.

Parameter	Value	
Vector length	20	
Reused data set size	20	
I/O IRAMs	14 I + 1 O + 1 internal	
ALU	20	
BREG	26 (8 defined + 18 route)	
FREG	28(4 defined + 24 route)	
Data flow graph width	14	
Data flow graph height	6 (without routing and balancing)	
Configuration cycles (simulated)	configuration	2633
	preloads	10*3*7*5 1050
		10*7*15 1050
	cycles	(k==0) 112 +
		(k==1) 100 +
		(k==7) 100
		* 10 ≈ 3120
	sum	7853

5.4 Viterbi Encoder

5.4.1 Original Code

Source Code:

```
/* C-language butterfly */
#define BFLY(i) {\
unsigned char metric,m0,m1,decision;\
    metric = ((Branchtab29_1[i] ^ sym1) +
              (Branchtab29_2[i] ^ sym2) + 1)/2;\
    m0 = vp->old_metrics[i] + metric;\
    m1 = vp->old_metrics[i+128] + (15 - metric);\
    decision = (m0-m1) >= 0;\
    vp->new_metrics[2*i] = decision ? m1 : m0;\
    vp->dp->w[i/16] |= decision << ((2*i)&31);\
    m0 -= (metric+metric-15);\
    m1 += (metric+metric-15);\
    decision = (m0-m1) >= 0;\
    vp->new_metrics[2*i+1] = decision ? m1 : m0;\
    vp->dp->w[i/16] |= decision << ((2*i+1)&31);\
}

int update_viterbi29(void *p,unsigned char sym1,unsigned char sym2){
  int i;
  struct v29 *vp = p;
  unsigned char *tmp;
  int normalize = 0;

  for(i=0;i<8;i++)
    vp->dp->w[i] = 0;

  for(i=0;i<128;i++)
    BFLY(i);

  /* Renormalize metrics */
  if(vp->new_metrics[0] > 150){
    int i;
    unsigned char minmetric = 255;

    for(i=0;i<64;i++)
      if(vp->new_metrics[i] < minmetric)
      minmetric = vp->new_metrics[i];
    for(i=0;i<64;i++)
      vp->new_metrics[i] -= minmetric;
    normalize = minmetric;
  }

  vp->dp++;
  tmp = vp->old_metrics;
  vp->old_metrics = vp->new_metrics;
  vp->new_metrics = tmp;

  return normalize;
}
```

5.4.2 Interprocedural Optimizations and Scalar Transformations

Since no inline-able function calls are present, no interprocedural code movement is done.

After expression simplification, strength reduction, SSA renaming, copy coalescing and idiom recognition, the code looks like (statements reordered for convenience). Note that idiom recognition will find the combination of *min()* and use of the comparison result for *decision* and *_decision*. However the resulting computation cannot be expressed in C, so we describe it as a comment:

```
int update_viterbi29(void *p,unsigned char sym1,unsigned char sym2){
  int i;
  struct v29 *vp = p;
  unsigned char *tmp;
  int normalize = 0;

  char *_vpdpw_= vp->dp->w;
  for(i=0;i<8;i++)
    *_vpdpw_++ = 0;

  char *_bt29_1= Branchtab29_1;
  char *_bt29_2= Branchtab29_2;
  char *_vpom0= vp->old_metrics;
  char *_vpom128= vp->old_metrics+128;
  char *_vpnm= vp->new_metrics;
  char *_vpdpw= vp->dp->w;

  for(i=0;i<128;i++) {
    unsigned char metric,_tmp, m0,m1,_m0,_m1, decision,_decision;

    metric = ((*_bt29_1++ ^ sym1) +
              (*_bt29_2++ ^ sym2) + 1)/2;
    _tmp= (metric+metric-15);
    m0 = *_vpom++ + metric;
    m1 = *_vpom128++ + (15 - metric);
    _m0 = m0 - _tmp;
    _m1 = m1 + _tmp;
    // decision = m0 >= m1;
    // _decision = _m0 >= _m1;
    *_vpnm++ = min(m0,m1);          // =  decision ? m1 : m0
    *_vpnm++ = min(_m0,_m1);        // = _decision ? _m1 : _m0
    _vpdpw[i >> 4] |= ( m0 >=  m1) /* decision*/ << ((2*i) & 31)
                    | (_m0 >= _m1) /*_decision*/ << ((2*i+1)&31);
}

/* Renormalize metrics */
if(vp->new_metrics[0] > 150){
  int i;
  unsigned char minmetric = 255;

  char *_vpnm= vp->new_metrics;
  for(i=0;i<64;i++)
    minmetric = min(minmetric, *vpnm++);

  char *_vpnm= vp->new_metrics;
  for(i=0;i<64;i++)
    *vpnm++ -= minmetric;
  normalize = minmetric;
}
```

```
  vp->dp++;
  tmp = vp->old_metrics;
  vp->old_metrics = vp->new_metrics;
  vp->new_metrics = tmp;

  return normalize;
}
```

5.4.3 Initialization

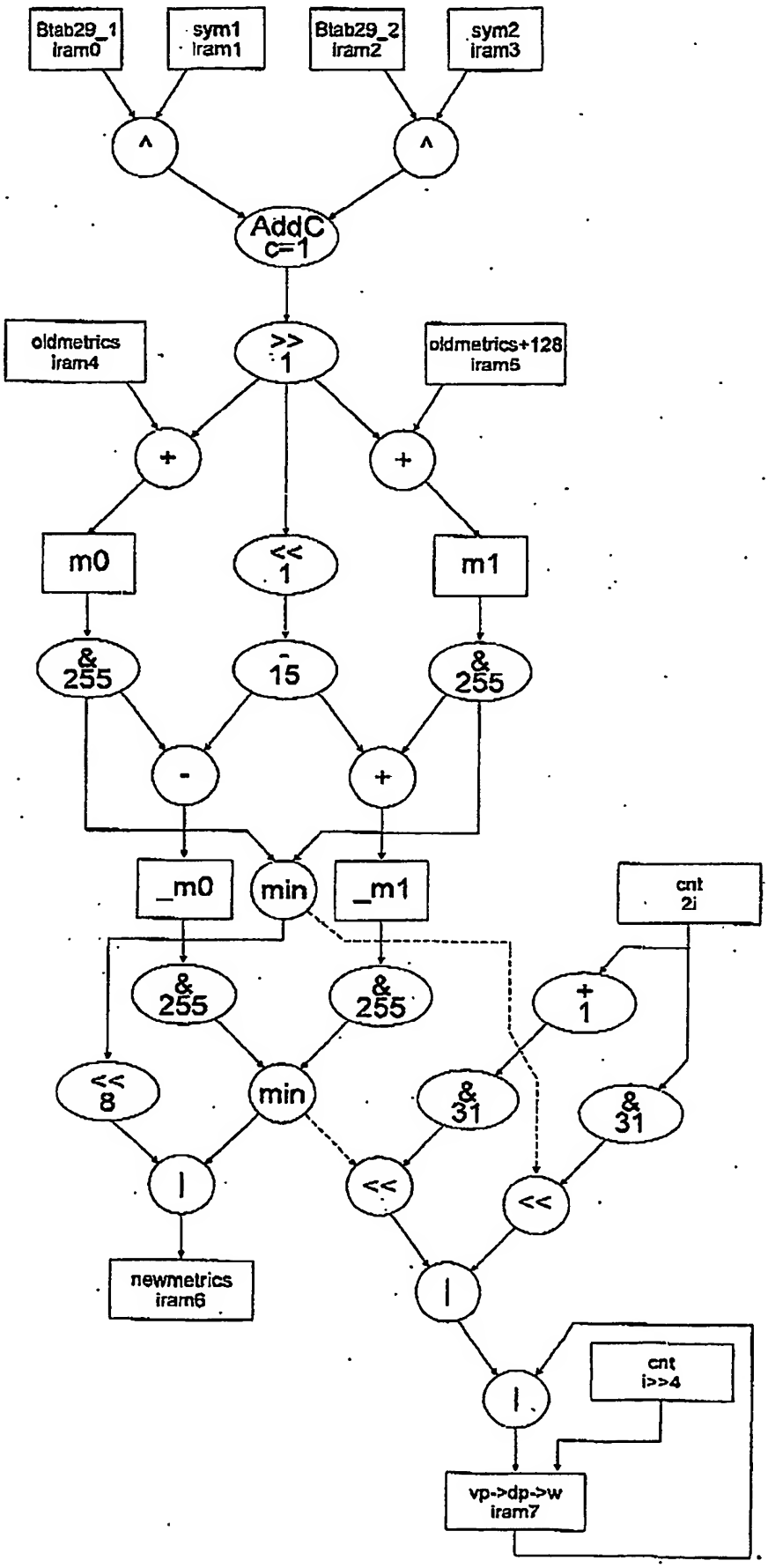
The first loop (setting vp->dp->w[0..7] to zero) is most efficiently executed on the RISC.

5.4.4 Butterfly Loop

The second loop (with the *BFLY()* macro expanded) is of interest for the XPP compiler and needs further examination:

```
  char *iram0= Branchtab29_1;        // XPPPreload(0, Branchtab29_1, 128/4);
  char *iram2= Branchtab29_2;        // XPPPreload(2, Branchtab29_2, 128/4);
  char *iram4= vp->old_metrics;      // XPPPreload(4, vp->old_metrics,    128/4);
  char *iram5= vp->old_metrics+128; // XPPPreload(5, vp->old_metrics+128,128/4);
 short *iram6= vp->new_metrics;     // XPPPreload(6, vp->new_metrics, 128/2);
  unsigned long *iram7= vp->dp->w;  // XPPPreload(7, vp->dp->w, 8);
// sym1 & sym2 are in IRAM 1 & 3

  for(i=0;i<128;i++) {
    unsigned char metric,_tmp, m0,m1,_m0,_m1

    metric = ((*iram0++ ^ sym1) +
              (*iram1++ ^ sym2) + 1)/2;
    _tmp= (metric << 1) -15;
    m0 = *iram2++ + metric;
    m1 = *iram3++ + (15 - metric);
    _m0 = m0 - _tmp;
    _m1 = m1 + _tmp;
    // assuming big endian; little endian has the shift on the latter min()
    *iram6++ = (min(m0,m1) << 8) | min(_m0,_m1);
    *iram7[i >> 4] |= ( m0 >=  m1) << ((2*i) & 31)
                    | (_m0 >= _m1) << ((2*i+1)&31);
  }
```

The data flow graph is as follows (for now ignoring the fact, that the IRAM accesses are mostly char accesses). The solid lines represent data flow, while the dashed lines represent event flow:


Btab29_1
iram0
sym1
iram1
Btab29_2
iram2
sym2
iram3
^
^
AddC
c=1
oldmetrics
iram4
>>
1
oldmetrics+128
iram5
+
+
m0
<<
1
m1
&
255
-15
&
255
-
+
_m0
min
_m1
cnt
2i
&
255
&
255
+
1
<<
8
min
&
31
&
31
|
<<
<<
newmetrics
iram6
|
cnt
i>>4
|
vp->dp->w
iram7

Parameter	Value
Vector length	128
Reused data set size	-
I/O IRAMs	6I+2O
ALU	25
BREG	few
FREG	few
Data flow graph width	4
Data flow graph height	11
Configuration cycles	11+128

We immediately see some problems: IRAM7 is fully busy reading and rewriting the same address sixteen times. Loop tiling to a tile size of sixteen gives the *redundant load store elimination* a chance to read the value once and accumulate the bits in the temporary, writing the value to the IRAM at the end of this inner loop. Loop Fusion with the initialization loop then allows propagation of the zero values set in the first loop to the reads of vp->dp->w[i] (IRAM7), eliminating the first loop altogether. Loop tiling with a tile size of 16 also eliminates the *& 31* expressions for the shift values: Since the new inner loop only runs from 0 to 16, the value range analysis now finds that the *& 31* expression is not limiting the value range any further.

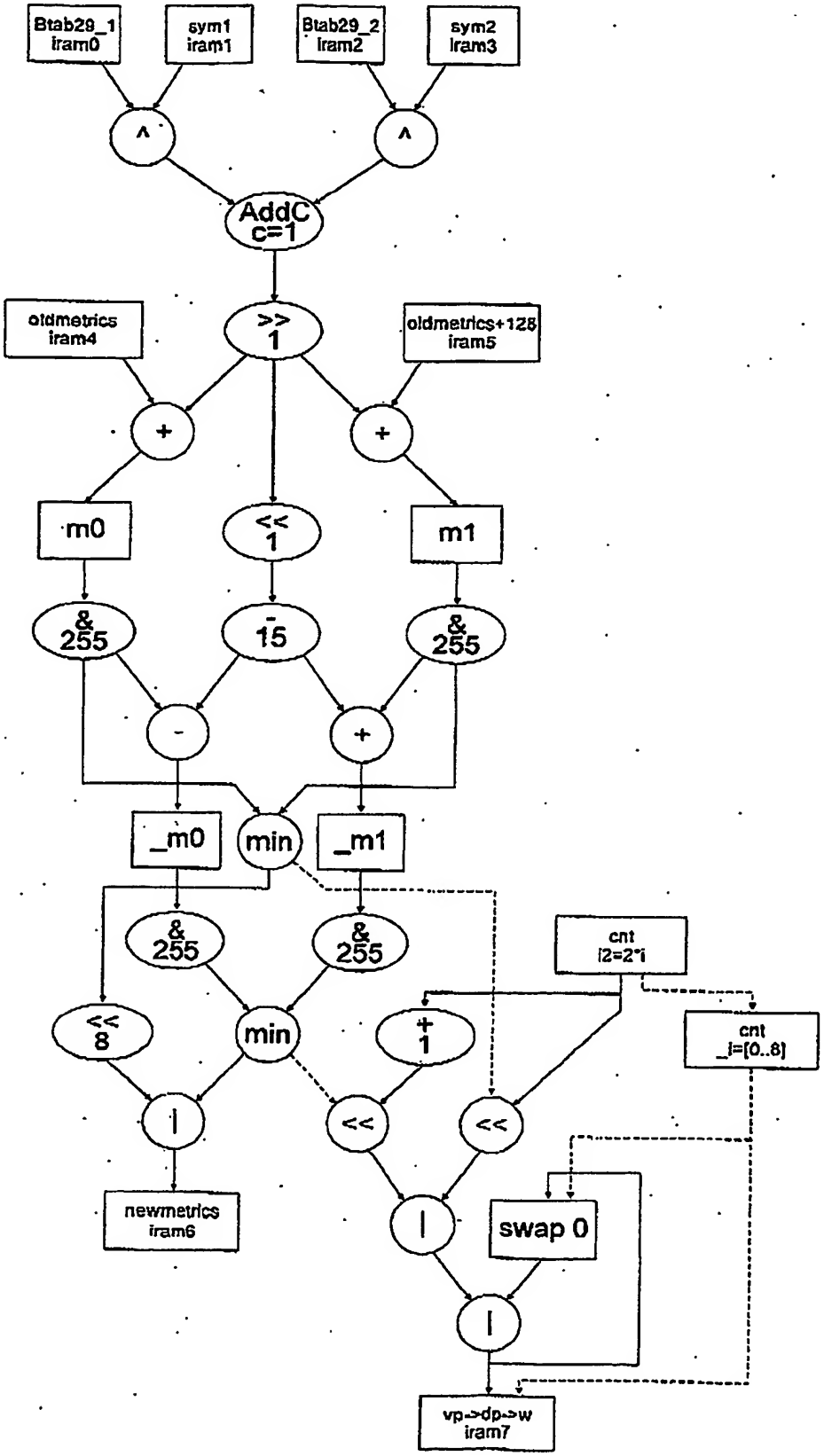
All remaining input IRAMs are character (8 bit) based. So we need split networks to split the 32-bit stream into four 8-bit streams which are then merged. This adds 3 shifts, 3 ands and 3 merges for every character IRAM. The merges could be eliminated, when unrolling the loop body. However, unrolling is limited to unrolling twice due to ALU availability as well as due to the fact, that IRAM6 is already 16 bit based: unrolling once requires a *shift by 16* and an *or* to write 32 bits in every cycle; unrolling further cannot increase pipeline throughput any more. So the body is only unrolled once, eliminating one layer of merges. This yields two separate pipelines, that each handle two eight bit slices of the 32-bit value from the IRAM, serialized by merges.

The modified code now looks like (unrolling and splitting omitted for simplicity):

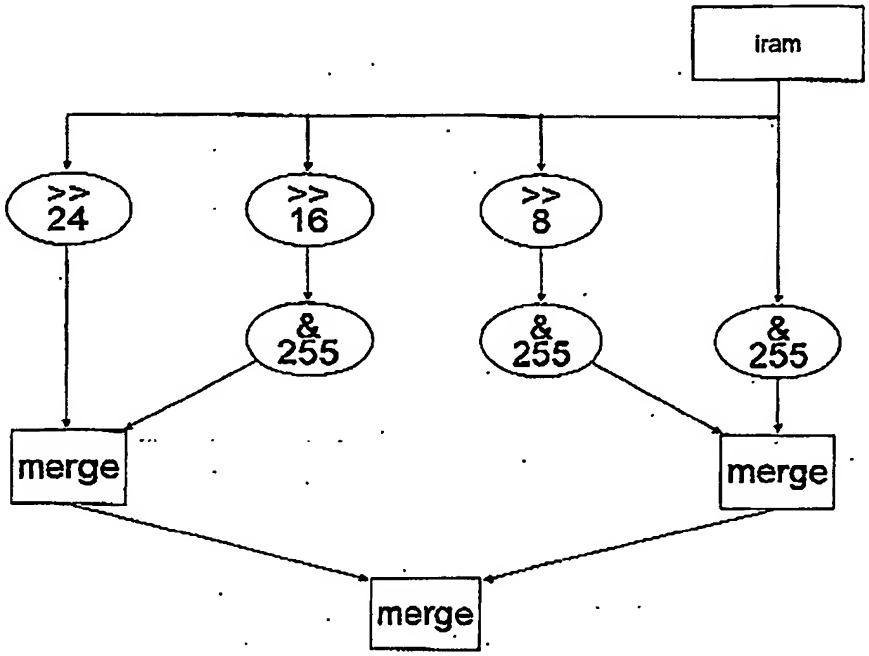
```
  char *iram0= Branchtab29_1;       // XPPPreload(0, Branchtab29_1, 128/4);
  char *iram2= Branchtab29_2;       // XPPPreload(2, Branchtab29_2, 128/4);
  char *iram4= vp->old_metrics;     // XPPPreload(4, vp->old_metrics,    128/4);
  char *iram5= vp->old_metrics+128; // XPPPreload(5, vp->old_metrics+128,128/4);
  short *iram6= vp->new_metrics;    // XPPPreload(6, vp->new_metrics, 128/2);
  unsigned long *iram7= vp->dp->w;  // XPPPreload(7, vp->dp->w, 8);
// sym1 & sym2 are in IRAM 1 & 3

  for(_i=0;_i<8;_i++) {
    rlse= 0;
    for(i2=0;i2<32;i2+=2) {
      unsigned char metric,_tmp, m0,m1,_m0,_m1;

      metric = ((*iram0++ ^ sym1) +
                (*iram1++ ^ sym2) + 1)/2;
      _tmp= (metric << 1) -15;
      m0 = *iram2++ + metric;
      m1 = *iram3++ + (15 - metric);
      _m0 = m0 - _tmp;
      _m1 = m1 + _tmp;
      *iram6++ = (min(m0,m1) << 8) | min(_m0,_m1);
      rlse = rlse | ( m0 >=  m1) << i2
                  | (_m0 >= _m1) << (i2+1);
    }
```

```
    *iram7++ = rlse;
}
```

The modified data flow graph (unrolling and splitting omitted for simplicity):



And here the splitting network for one IRAM: the bottom most level merge is omitted for each level of unrolling.



Parameter	Value
Vector length	128
Reused data set size	-
I/O IRAMs	6I+2O
ALU	2*24+4*3(split)+2(join)= 62
BREG	few
FREG	few
Data flow graph width	4
Data flow graph height	11+3(split)
Configuration cycles	14+64

5.4.5 Re-Normalization:

The Normalization consists of a loop scanning the input for the minimum and a second loop that subtracts the minimum from all elements. There is a data dependency between all iterations of the first loop and all iterations of the second loop. Therefore the two loops cannot be merged or pipelined. They will be handled individually.

Minimum Search

The third loop is a minimum search on a byte array.

```
char *iram0 = vp->new_metrics;  // XPPPreload(0, vp->new_metrics, 64/4);
for(i=0;i<64;i++)
  minmetric = min(minmetric, *iram0++);
```

Parameter	Value
Vector length	64
Reused data set size	-
I/O IRAMs	1+1
ALU	1
BREG	0
FREG	0
Data flow graph width	1
Data flow graph height	1
Configuration cycles	64

Reduction recognition eliminates the dependence for *minmetric* enabling a four-times unroll to utilize the IRAM width of 32 bits. A split network has to be added to separate the 8 bit streams using 3 SHIFT and 3 AND operations. Tree balancing re-distributes the *min()* operations to minimize the tree height.

```
char *iram0 = vp->new_metrics;  // XPPPreload(0, vp->new_metrics, 16);
for(i=0;i<16;i++)
  minmetric = min(minmetric, min( min(*iram0++, *iram0++),
                                  min(*iram0++, *iram0++) ));
```

Parameter	Value
Vector length	16
Reused data set size	-
I/O IRAMs	1I+1O
ALU	4*min
BREG	3*shln+3*shrn
FREG	0
Data flow graph width	4
Data flow graph height	5
Configuration cycles	5+16

Reduction recognition again eliminates the loop carried dependence for *minmetric*, enabling loop tiling and then unroll and jam to increase parallelism; the maximum for the tiling size is 16 IRAMs / 2 IRAMS = 8. Constant propagation and tree rebalancing reduces the dependence height of the final merging expression:

```
char *iram0= vp->new_metrics;     // XPPPreload(0, vp->new_metrics, 2);
char *iram1= vp->new_metrics+8;   // XPPPreload(1, vp->new_metrics+8, 2);
char *iram2= vp->new_metrics+16;  // XPPPreload(2, vp->new_metrics+16, 2);
char *iram3= vp->new_metrics+24;  // XPPPreload(3, vp->new_metrics+24, 2);
char *iram4= vp->new_metrics+32;  // XPPPreload(4, vp->new_metrics+32, 2);
char *iram5= vp->new_metrics+40;  // XPPPreload(5, vp->new_metrics+40, 2);
char *iram6= vp->new_metrics+48;  // XPPPreload(6, vp->new_metrics+48, 2);
char *iram7= vp->new_metrics+56;  // XPPPreload(7, vp->new_metrics+56, 2);
```

```
for(i=0;_i<2;i++) {
  minmetric0  = min(minmetric0 , min( min(*iram0++, *iram0++),
                                      min(*iram0++, *iram0++) ));
  minmetric1  = min(minmetric1 , min( min(*iram1++, *iram1++),
                                      min(*iram1++, *iram1++) ));
  minmetric2  = min(minmetric2 , min( min(*iram2++, *iram2++),
                                      min(*iram2++, *iram2++) ));
  minmetric3  = min(minmetric3 , min( min(*iram3++, *iram3++),
                                      min(*iram3++, *iram3++) ));
  minmetric4  = min(minmetric4 , min( min(*iram4++, *iram4++),
                                      min(*iram4++, *iram4++) ));
  minmetric5  = min(minmetric5 , min( min(*iram5++, *iram5++),
                                      min(*iram5++, *iram5++) ));
  minmetric6  = min(minmetric6 , min( min(*iram6++, *iram6++),
                                      min(*iram6++, *iram6++) ));
  minmetric7  = min(minmetric7 , min( min(*iram7++, *iram7++),
                                      min(*iram7++, *iram7++) ));
}
minmetric = min( min((min(minmetric_0, minmetric_1),
                      min(minmetric_2, minmetric_3)),
                 min((min(minmetric_4, minmetric_5),
                      min(minmetric_6, minmetric_7));
```

Parameter	Value
Vector length	2
Reused data set size	-
I/O IRAMs	8I+1O
ALU	8*4*min = 32
BREG	8*(3*shln+3*shrn)= 48
FREG	0
Data flow graph width	8*4= 32
Data flow graph height	5
Configuration cycles	8+2

Re-Normalization

The fourth loop subtracts the minimum of the third loop from each element in the array. The read-modify-write operation has to be broken up into two IRAMs. Otherwise the IRAM ports will limit throughput.

```
char *iram0= vp->new_metrics;   // XPPPreload      (0, vp->new_metrics, 64/4)
char *iram1= vp->new_metrics;   // XPPPreloadClean(1, vp->new_metrics, 64/4)
for(i=0;i<64;i++)
  *iram1++ = *iram0++ - minmetric;
```

Parameter	Value
Vector length	64
Reused data set size	-
I/O IRAMs	2I+1O
ALU	.I
BREG	0
FREG	0
Data flow graph width	1
Data flow graph height	1
Configuration cycles	64

There are no loop carried dependencies. Since the data size is bytes, the inner loop can be unrolled four times without exceeding the IRAM bandwidth requirements. Networks splitting the 32-bit stream into 4 8-bit streams and re-joining the individual results to a common 32-bit result stream are inserted.

```
char *iram0= vp->new_metrics;    // XPPPreload      (0, vp->new_metrics, 16)
char *iram1= vp->new_metrics;    // XPPPreloadClean(1, vp->new_metrics, 16)
for(i=0;i<16;i++) {
  *iram1++ = *iram0++ - minmetric;
  *iram1++ = *iram0++ - minmetric;
  *iram1++ = *iram0++ - minmetric;
  *iram1++ = *iram0++ - minmetric;
}
```

Parameter	Value
Vector length	16
Reused data set size	-
I/O IRAMs	2I+1°
ALU	4*4(sub) = 16
BREG	6*shln+6*shrn= 12
FREG	0
Data flow graph width	4
Data flow graph height	5
Configuration cycles	2(split)+4*1(sub)+2(join)= 8

Unroll and jam can be applied after loop tiling, in analogy to the third loop, but loop tiling is now limited by the BREGs used by the split and join networks. The computed tiling size (unroll factor) is 64 BREGs/12 BREGs = 5, which is replaced by 4, since the same throughput is achieved with less overhead.

```
char *iram0= vp->new_metrics;     // XPPPreload      (0,vp->new_metrics,   4)
char *iram1= vp->new_metrics;     // XPPPreloadClean(1,vp->new_metrics,   4)
char *iram2= vp->new_metrics+16;  // XPPPreload      (2,vp->new_metrics+16,4)
char *iram3= vp->new_metrics+16;  // XPPPreloadClean(3,vp->new_metrics+16,4)
char *iram4= vp->new_metrics+32;  // XPPPreload      (4,vp->new_metrics+32,4)
char *iram5= vp->new_metrics+32;  // XPPPreloadClean(5,vp->new_metrics+32,4)
char *iram6= vp->new_metrics+48;  // XPPPreload      (6,vp->new_metrics+48,4)
char *iram7= vp->new_metrics+48;  // XPPPreloadClean(7,vp->new_metrics+48,4)
```

```
for(i=0;i<4;i++) {
  *iram1++ = *iram0++ - minmetric;              // first pipeline
  *iram1++ = *iram0++ - minmetric;
  *iram1++ = *iram0++ - minmetric;
  *iram1++ = *iram0++ - minmetric;
  *iram3++ = *iram2++ - minmetric;              // second pipeline
  *iram3++ = *iram2++ - minmetric;
  *iram3++ = *iram2++ - minmetric;
  *iram3++ = *iram2++ - minmetric;
  *iram5++ = *iram4++ - minmetric;              // third pipeline
  *iram5++ = *iram4++ - minmetric;
  *iram5++ = *iram4++ - minmetric;
  *iram5++ = *iram4++ - minmetric;
  *iram7++ = *iram6++ - minmetric;              // fourth pipeline
  *iram7++ = *iram6++ - minmetric;
  *iram7++ = *iram6++ - minmetric;
  *iram7++ = *iram6++ - minmetric;
}
```

Parameter	Value
Vector length	4
Reused data set size	-
I/O IRAMs	5I+4O
ALU	4*(6(split)+4(sub)+6(join)) = 64
BREG	4*(6*shln+6*shrn)= 48
FREG	0
Data flow graph width	16
Data flow graph height	1
Configuration cycles	2(split)+4*1(sub)+2(join)= 8

5.4.6 Final Code

Finally we arrive at the following code:

```
int update_viterbi29(void *p,unsigned char sym1,unsigned char sym2){
  int i;
  struct v29 *vp = p;
  unsigned char *tmp;
  int normalize = 0;

  // initialization loop eliminated
  // for(i=0;i<8;i++)
  //   vp->dp->w[i] = 0;

  // Configuration for butterfly loop
  char *iram0= Branchtab29_1;       // XPPPreload(0, Branchtab29_1, 128/4);
  char *iram2= Branchtab29_2;       // XPPPreload(2, Branchtab29_2, 128/4);
  char *iram4= vp->old_metrics;     // XPPPreload(4, vp->old_metrics,    128/4);
  char *iram5= vp->old_metrics+128; // XPPPreload(5, vp->old_metrics+128,128/4);
  short *iram6= vp->new_metrics;    // XPPPreload(6, vp->new_metrics, 128/2);
  unsigned long *iram7= vp->dp->w;  // XPPPreload(7, vp->dp->w, 8);
// sym1 & sym2 are in IRAM 1 & 3
```

```
  for(_i=0;_i<8;_i++) {
  rlse= 0;
  for(i2=0;i<32;i2+=2) { // unrolled once
    unsigned char metric,_tmp, m0,m1,_m0,_m1

     metric = ((*iram0++ ^ sym1) +
               (*iram1++ ^ sym2) + 1)/2;
     _tmp= (metric << 1) -15;
     m0 = *iram2++ + metric;
     m1 = *iram3++ + (15 - metric);
     _m0 = m0 - _tmp;
     _m1 = m1 + _tmp;
     *iram6++ = (min(m0,m1) << 8) | min(_m0,_m1);
     rlse = rlse | ( m0 >=  m1) << i2
                 | (_m0 >= _m1) << (i2+1);
   }
   *iram7++ = rlse;
 }

  /* Renormalize metrics */
  if(vp->new_metrics[0] > 150){
    int i;

// Configuration for loop 3
    char *iram0= vp->new_metrics;     // XPPPreload(0, vp->new_metrics, 8);
    char *iram1= vp->new_metrics+8;   // XPPPreload(1, vp->new_metrics+8, 8);
    char *iram2= vp->new_metrics+16;  // XPPPreload(2, vp->new_metrics+16, 8);
    char *iram3= vp->new_metrics+24;  // XPPPreload(3, vp->new_metrics+24, 8);
    char *iram4= vp->new_metrics+32;  // XPPPreload(4, vp->new_metrics+32, 8);
    char *iram5= vp->new_metrics+40;  // XPPPreload(5, vp->new_metrics+40, 8);
    char *iram6= vp->new_metrics+48;  // XPPPreload(6, vp->new_metrics+48, 8);
    char *iram7= vp->new_metrics+56;  // XPPPreload(7, vp->new_metrics+56, 8);
    for(i=0;_i<2;i++) {
      minmetric0  = min(minmetric0 , min( min(*iram0++, *iram0++),
                                          min(*iram0++, *iram0++) ));
      minmetric1  = min(minmetric1 , min( min(*iram1++, *iram1++),
                                          min(*iram1++, *iram1++) ));
      minmetric2  = min(minmetric2 , min( min(*iram2++, *iram2++),
                                          min(*iram2++, *iram2++) ));
      minmetric3  = min(minmetric3 , min( min(*iram3++, *iram3++),
                                          min(*iram3++, *iram3++) ));
      minmetric4  = min(minmetric4 , min( min(*iram4++, *iram4++),
                                          min(*iram4++, *iram4++) ));
      minmetric5  = min(minmetric5 , min( min(*iram5++, *iram5++),
                                          min(*iram5++, *iram5++) ));
      minmetric6  = min(minmetric6 , min( min(*iram6++, *iram6++),
                                          min(*iram6++, *iram6++) ));
      minmetric7  = min(minmetric7 , min( min(*iram7++, *iram7++),
                                          min(*iram7++, *iram7++) ));
    }
    minmetric = min( min((min(minmetric_0, minmetric_1),
                          min(minmetric_2, minmetric_3)),
                     min((min(minmetric_4, minmetric_5),
                          min(minmetric_6, minmetric_7));
// minmetric is written to the output IRAM
```

```
// Configuration for loop 4, minmetric is in an input IRAM
    char *iram0= vp->new_metrics;     // XPPPreload      (0,vp->new_metrics,   4)
    char *iram1= vp->new_metrics;     // XPPPreloadClean(1,vp->new_metrics,   4)
    char *iram2= vp->new_metrics+16; // XPPPreload      (2,vp->new_metrics+16,4)
    char *iram3= vp->new_metrics+16; // XPPPreloadClean(3,vp->new_metrics+16,4)
    char *iram4= vp->new_metrics+32; // XPPPreload      (4,vp->new_metrics+32,4)
    char *iram5= vp->new_metrics+32; // XPPPreloadClean(5,vp->new_metrics+32,4)
    char *iram6= vp->new_metrics+48; // XPPPreload      (6,vp->new_metrics+48,4)
    char *iram7= vp->new_metrics+48; // XPPPreloadClean(7,vp->new_metrics+48,4)
    for(i=0;i<4;i++) {
      *iram1++ = *iram0++ - minmetric;    // first pipeline
      *iram1++ = *iram0++ - minmetric;
      *iram1++ = *iram0++ - minmetric;
      *iram1++ = *iram0++ - minmetric;
      *iram3++ = *iram2++ - minmetric;    // second pipeline
      *iram3++ = *iram2++ - minmetric;
      *iram3++ = *iram2++ - minmetric;
      *iram3++ = *iram2++ - minmetric;
      *iram5++ = *iram4++ - minmetric;    // third pipeline
      *iram5++ = *iram4++ - minmetric;
      *iram5++ = *iram4++ - minmetric;
      *iram5++ = *iram4++ - minmetric;
      *iram7++ = *iram6++ - minmetric;    // fourth pipeline
      *iram7++ = *iram6++ - minmetric;
      *iram7++ = *iram6++ - minmetric;
      *iram7++ = *iram6++ - minmetric;
    }
    normalize = minmetric;
  }

  vp->dp++;
  tmp = vp->old_metrics;
  vp->old_metrics = vp->new_metrics;
  vp->new_metrics = tmp;

  return normalize;
}
```

Performance Considerations

In this example we do not have a high data locality. Every input data item is read exactly once. Only in the case of re-normalization, the *new_metric* array is re-read and re-written. To fully utilize the PAE array, loop tiling was used – in conjunction with reduction recognition to break dependencies using algebraic identities. In some cases (minimum search) this leads to extremely short vector lengths. This does not hurt as it still does reduce the running time of the configuration and the transfer time from the top of the memory hierarchy to the IRAMs stays the same. The vector length could be increased if the outer loop that calls the function was known – the additional data could be used to increase the fill grade of the IRAMs by unrolling the outer loop, keeping the vector length longer. This would further increase configuration performance by reducing overall pipeline setup times.

Performance of XPP for this example is compared to a hypothetical superscalar RISC-architecture. We

Operation	Cycles	Bfly Setup	Butterfly	Min Setup	Min Search	Norm Setup	Normalize
ADRCOMP	1	6	7		1		
LD/ST	2	5	8	2		1	2
LDI	1	3	4	1		1	
MOVE	1		4		1		
BITOP	1		10				
ADD/SUB	1		20		3	1	3
MULT	2	2					
CJMP	3		3		2		1
Cycles		23	70	5	11	4	10
Count		1	128	1	64	1	64
Issue Width	2						
Total Cycles		12	4480	3	352	2	320

Est. RISC cycles
5168 RISC Cycles

Est. XPP cycles

assume an average issue width of two which means that the RISC on average executes two operations in parallel. The estimate is achieved by counting instructions for the source code in 5.4.2.

5.5 MPEG2 encoder/decoder

5.5.1 Quantization / Inverse Quantization (quant.c)

The quantization file contains routines for quantization and inverse quantization of 8x8 macro blocks. These functions differ for intra and non-intra blocks and furthermore the encoder distinguishes between MPEG1 and MPEG2 inverse quantization.

This gives a total of 6 functions, which are all candidates for function inlining, since they do not use the XPP capacity by far.

Since all functions have the same layout (some checks, one main loop running over the macro block quantizing with a quantization matrix), we concentrate on "iquant_intra", the inverse quantization of intra-blocks, since it contains all elements found in the other procedures (The non_intra quantization loop bodies are more complicated, but add no compiler complexity). In the source code the mpeg1 part is already inlined, which is straightforward since the function is statically defined and contains no function calls itself. Therefore the compiler inlines it and dead function elimination removes the whole definition.

Original Code

```
void iquant_intra(src,dst,dc_prec,quant_mat,mquant)
short *src, *dst;
int dc_prec;
unsigned char *quant_mat;
int mquant;
{
  int i, val, sum;

  if (mpeg1) {
    dst[0] = src[0] << (3-dc_prec);
    for (i=1; i<64; i++)
    {
      val = (int)(src[i]*quant_mat[i]*mquant)/16;

      /* mismatch control */
      if ((val&1)==0 && val!=0)
        val+= (val>0) ? -1 : 1;

      /* saturation */
      dst[i] = (val>2047) ? 2047 : ((val<-2048) ? -2048 : val);
  }
  else
  {
    sum = dst[0] = src[0] << (3-dc_prec);
    for (i=1; i<64; i++)
    {
      val = (int)(src[i]*quant_mat[i]*mquant)/16;
      sum+= dst[i] = (val>2047) ? 2047 : ((val<-2048) ? -2048 : val);
    }

    /* mismatch control */
    if ((sum&1)==0)
      dst[63]^= 1;
  }
```

```
}
```

Interprocedural Optimizations

Analysing the loop bodies shows that they easily fit to the XPP and do not use the maximum of resources by far. The function is called three times from module putseq.c. With inter-module function inlining the code for the function call disappears and is replaced with the function Therefore it reads

```
for (k=0; k<mb_height*mb_width; k++) {
  if (mbinfo[k].mb_type & MB_INTRA)
    for (j=0; j<block_count; j++)
      if (mpeg1) {
        blocks[k*block_count+j][0] = blocks[k*block_count+j][0] <<
                                     (3-dc_prec);
        for (i=1; i<64; i++) {
          val = (int)( blocks[k*block_count+j][i]*intra_q[i]*mquant)/16;
          ..............
        }
      } else {
        sum = blocks[k*block_count+j][0] = blocks[k*block_count+j][0] <<
                                           (3-dc_prec);
        for (i=1; i<64; i++) {
          val = (int)( blocks[k*block_count+j][i]* intra_q [i]*mquant)/16;
          ..............
        }
      }
} else {
  ..............
}
```

Basic transformations

Since global mpeg1 does not change within the loop unswitching moves the control statement outside the j loop and produces two loop nests.

```
for (k=0; k<mb_height*mb_width; k++) {
  if (mbinfo[k].mb_type & MB_INTRA)
    if (mpeg1)
      for (j=0; j<block_count; j++) {
        blocks[k*block_count+j][0] = blocks[k*block_count+j][0] <<
                                     (3-dc_prec);
        for (i=1; i<64; i++) {
          val = (int)( blocks[k*block_count+j][i]*intra_q[i]*mquant)/16;
          ..............
        }
      }
    else
      for (j=0; j<block_count; j++) {
        sum = blocks[k*block_count+j][0] = blocks[k*block_count+j][0] <<
                                           (3-dc_prec);
        for (i=1; i<64; i++) {
          val = (int)( blocks[k*block_count+j][i]* intra_q [i]*mquant)/16;
          ..................
        }
      }
}
```

Furthermore the following transformations are done:

- A peephole optimization reduces the divide by 16 to a right shift 4. This is essential since we do not consider loop bodies containing division for the XPP.
- Idiom recognition reduces the statement after the "saturation" comment to `dst[i] = min(max(val, -2048), 2047)`

Increasing parallelism

Now we want to increase parallelism. The j-i loop nest is a candidate for unroll-and-jam when the interprocedural value range analysis finds out that block_count can only get the values 6,8 or 12. Therefore it has a value range [6,12] with the additional attribute to be dividable by 2. Thus an unroll and jam with the factor 2 is applicable (the resource constraints would choose a bigger value). Since no loop carried dependencies exist, this transformation is safe.

It is to say that the source code contains a manually peeled first iteration. This peeling has been done because the value calculated for the first block value is completely different from the other iterations and the control statement in the loop would cause a major performance decrease on traditional processors. Although this does not prevent unroll-and-jam (because there are no dependencies between the peeled of first iteration and the rest of the loop), the transformation must be prepared to handle such cases.

After unroll and jam the source code looks like (only one of the nests showed and the peeled first iterations moved in front)

```
for (j=0; j<block_count; j+=2) {
  blocks[k*count+j][0] = blocks[k*count+j][0] << (3-dc_prec);
  blocks[k*count+j+1][0] = blocks[k*count+j+1][0] << (3-dc_prec);
  for (i=1; i<64; i++) {
    val = (int)(blocks[k*count+j][i]*intra_q[i]*mbinfo[k].mquant)>>4;

    /* mismatch control */
    if ((val&1)==0 && val!=0)
      val+= (val>0) ? -1 : 1;

    /* saturation */
    blocks[k*count+j][i] = min(max(val, -2048), 2047);

    val = (int)(blocks[k*count+j+1][i]*intra_q[i]*mbinfo[k].mquant)>>4;

    /* mismatch control */
    if ((val&1)==0 && val!=0)
      val+= (val>0) ? -1 : 1;

    /* saturation */
    blocks[k*count+j+1][i] = min(max(val, -2048), 2047);
  }
}
```

Further parallelism can be obtained by index set splitting. Normally used to break dependence cycles in the DDG, it can here be used to split the i-loop in two and let two sub-configurations[5] work on distinct blocks of data. Thus the i loop is split into 2 or more loops which work on different subsets of the data at the same time.

[5] sub-configuration is chosen as a working title for configurations which contains independent networks that do not interfere.

Handling the data types

In contrast to the FIR-Filter, edge detector and matrix multiplication benchmarks, which all use data types fitting perfectly to the XPP[6], the MPEG2 codec uses all data types commonly used on a processor for desktop applications. Written for the Intel x86 and comparable architectures, we must assume that the sizes of char, short and int are 8,16, and 32 respectively. Assuming that the XPP has a bit width of 32 we must take precautions for the smaller data types.

Therefore we split the stream of data packets with each packet containing 2 or 4 values of the shorter data type into 2 or 4 streams. If we have enough resources left, this will cause no performance penalty. Each of the divided streams is sent to its own calculation network; therefore in every cycle two short or four char values are handled. Nevertheless this causes an area penalty, because besides the split-merge elements, the whole data flow graph has to be duplicated as often as needed. Figure 63 shows how short values are handled. The packet is split into its hi- and lo part by shift operations and merged behind the calculation branches. The legality of this transformation is the same as with loop unrolling with an unrolling factor as big as the data type is smaller as the architecture data type.

Unfortunately this is not the end of the pole. The compiler further has to assure that every intermediate result which produces an over/under-flow for the shorter data type does the same with the bigger data type. Therefore it has to insert clipping operations which assure that the network calculates with real 16 or 8 bit value, respectively.

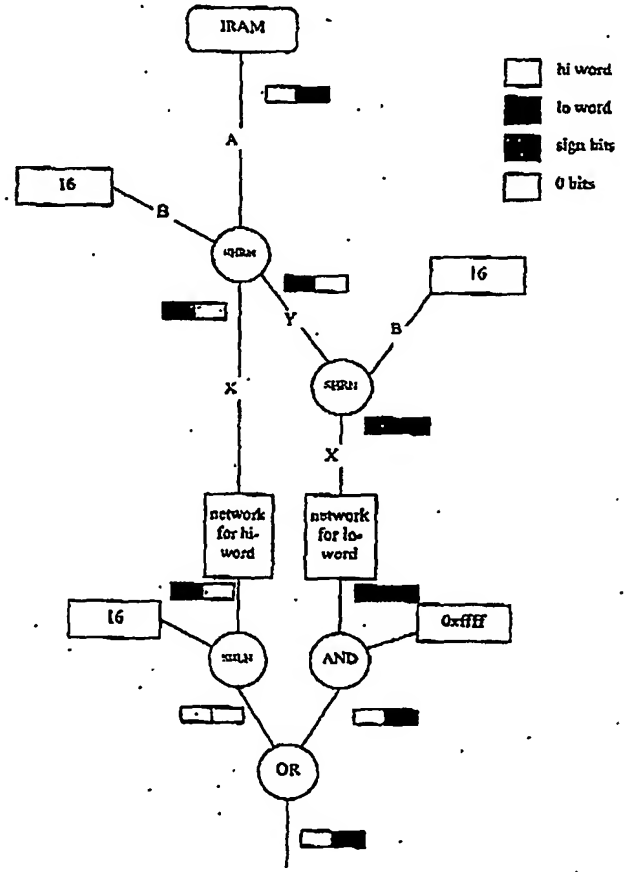


Figure 63 Splitting short values into two streams and merging them after the calculation. This method causes no performance penalty

[6] We assume that the size of int is chosen to be the XPP architecture data bit width. Everything else would not lead to any feasible result

If the configuration size does not allow the whole loop body to be duplicated or dependencies prevent this, we still have the possibility to merge the split values again. This of course causes a performance penalty to the previous solution, because the throughput is only one (short) value/cycle now. Figure 64 shows how the merge is done. Instead of streaming parallel through two networks the values are serialized and de-serialized again after the network.

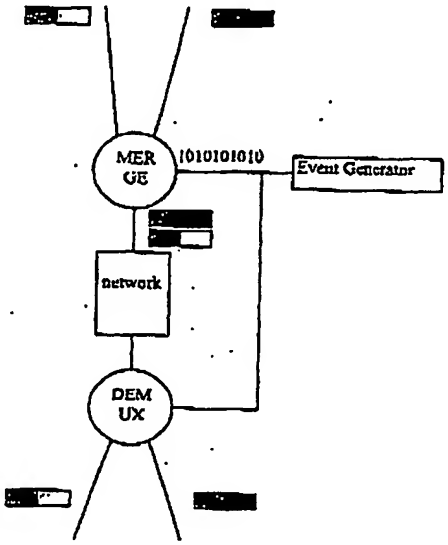


Figure 64 Merging the split values before the network. An event generator drives the merge and demux PAEs. This figure replaces the 2 black boxes labeled "network" in Figure 63

5.5.2 Inverse Discrete Cosine Transformation (idct.c)

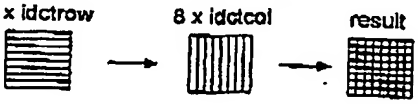
The idct-algorithm is used for the MPEG2 video decompression algorithm. It operates on 8x8 blocks of video images in their frequency representation and transforms them back into their original signal form. The MPEG2 decoder contains a transform-function that calls idct for all blocks of a frequency-transformed picture to restore the original image.

The idct function consists of two for-loops. The first loop calls idctrow - the second idctcol. Function inlining is able to eliminate the function calls within the entire loop nest structure so that the numeric code is not interrupted by function calls anymore. Another way to get rid of function calls between the loop nest is loop embedding that pushes loops from the caller into the callee.

Original Code (idct.c)

```
/* two dimensional inverse discrete cosine transform */
void idct(block)
short *block;
{
  int i;

  for (i=0; i<8; i++)
    idctrow(block+8*i);

  for (i=0; i<8; i++)
    idctcol(block+i);
}
```

The first loop changes the values of the block row by row. Afterwards the changed block is further transformed column by column. All rows have to be finished before any column processing can be started.



Dependency analysis detects true data dependencies between row processing and column processing. Therefore the processing of the columns has to be delayed until all rows are done. The innermost loop bodies idctrow and idctcol are nearly identical. They process numeric calculations on eight input values (column values in case of idctcol and row values in case of idctcol). Eight output values are calculated and written back (as column/row). Idctcol additionally applies clipping before the values are written back. This is why we concentrate on idctcol:

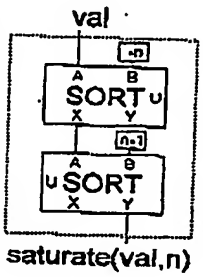
```
/* column (vertical) IDCT
 *
 *             7                         pi         1
 * dst[8*k] = sum c[l] * src[8*l] * cos( -- * ( k + - ) * l )
 *            l=0                        8         2
 *
 * where: c[0]    = 1/1024
 *        c[1..7] = (1/1024)*sqrt(2)
 */
static void idctcol(blk)
short *blk;
{
  int x0, x1, x2, x3, x4, x5, x6, x7, x8;


  /* shortcut */
  if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) |
        (x3 = blk[8*2]) | (x4 = blk[8*1]) | (x5 = blk[8*7]) |
        (x6 = blk[8*5]) | (x7 = blk[8*3])))
  {
     blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=blk[8*4]=blk[8*5]=
     blk[8*6]=blk[8*7]=iclp[(blk[8*0]+32)>>6];
     return;
  }

  x0 = (blk[8*0]<<8) + 8192;

  /* first stage */
  x8 = W7*(x4+x5) + 4;
  x4 = (x8+(W1-W7)*x4)>>3;
  x5 = (x8-(W1+W7)*x5)>>3;
  x8 = W3*(x6+x7) + 4;
  x6 = (x8-(W3-W5)*x6)>>3;
  x7 = (x8-(W3+W5)*x7)>>3;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2) + 4;
  x2 = (x1-(W2+W6)*x2)>>3;
  x3 = (x1+(W2-W6)*x3)>>3;
```

```
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;

    /* third stage */
    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181*(x4+x5)+128)>>8;
    x4 = (181*(x4-x5)+128)>>8;

    /* fourth stage */
    blk[8*0] = iclp[(x7+x1)>>14];
    blk[8*1] = iclp[(x3+x2)>>14];
    blk[8*2] = iclp[(x0+x4)>>14];
    blk[8*3] = iclp[(x8+x6)>>14];
    blk[8*4] = iclp[(x8-x6)>>14];
    blk[8*5] = iclp[(x0-x4)>>14];
    blk[8*6] = iclp[(x3-x2)>>14];
    blk[8*7] = iclp[(x7-x1)>>14];
}
```

W1 – W7 are macros for numeric constants that are substituted by the preprocessor. The iclp array is used for clipping the results to 8-bit values. It is fully defined by the init_idct function before idct is called the first time:

```
void init_idct()
{
  int i;

  iclp = iclip+512;
  for (i= -512; i<512; i++)
    iclp[i] = (i<-256) ? -256 : ((i>255) ? 255 : i);
}
```

A special kind of idiom recognition (function recognition) is able to replace the calculation of each array element by a compiler known function that can be realized efficiently on the XPP. If the compiler features whole program memory aliasing analysis it is able to replace all uses of the iclp array with the call of the compiler known function. Alternatively a developer can replace the iclp array accesses manually by the compiler known saturation function calls. The illustration shows a possible implementation for saturate(val,n) as NML schematic using two ALUs. In this case it is necessary to replace array accesses like iclp[i] by saturate(i,256).



saturate(val,n)

The /*shortcut*/ code in idctcol speeds column processing up if x1 to x7 is zero. This breaks the well-formed structure of the loop nest. The if-condition is not loop invariant and loop unswitching cannot be applied. But nonetheless - the code after shortcut handling is well suited for the XPP. It is possible to synthesize if-conditions for the XPP (speculative processing of both blocks plus selection based on condition) but this would just waste PAEs without any performance benefit. Therefore the /*shortcut*/ code in idctrow and idctcol has to be removed manually. The code snippet below shows the inlined version of the idctrow-loop with additional cache instructions for XPP control:

```
void idct(block)
short *block;
{
  int i;

  XPPPreload(IDCTROW_CONFIG); // Loop Invariant

  for (i=0; i<8; i++) {
      short *blk;
      int x0, x1, x2, x3, x4, x5, x6, x7, x8;
      blk = block+8*i;

      XPPPreload(0, blk, 8);

      XPPPreloadClean(1,blk,8); // IRAM1 is erased and assigned to blk

      XPPExecute(IDCTROW_CONFIG,IRAM(0),IRAM(1));

  }
  for (i=0; i<8; i++) {
      ...
  }
}
```

As the configuration of the XPP does not change during the loop execution invariant code motion has moved out XPPPreload(IDCTROW_CONFIG) from the loop.

NML Code Generation

Data Flow Graph

As idctcol is more complex due to clipping at the end of the calculations we decided to take idctcol as representative loop body for a presentation of the data flow graph.

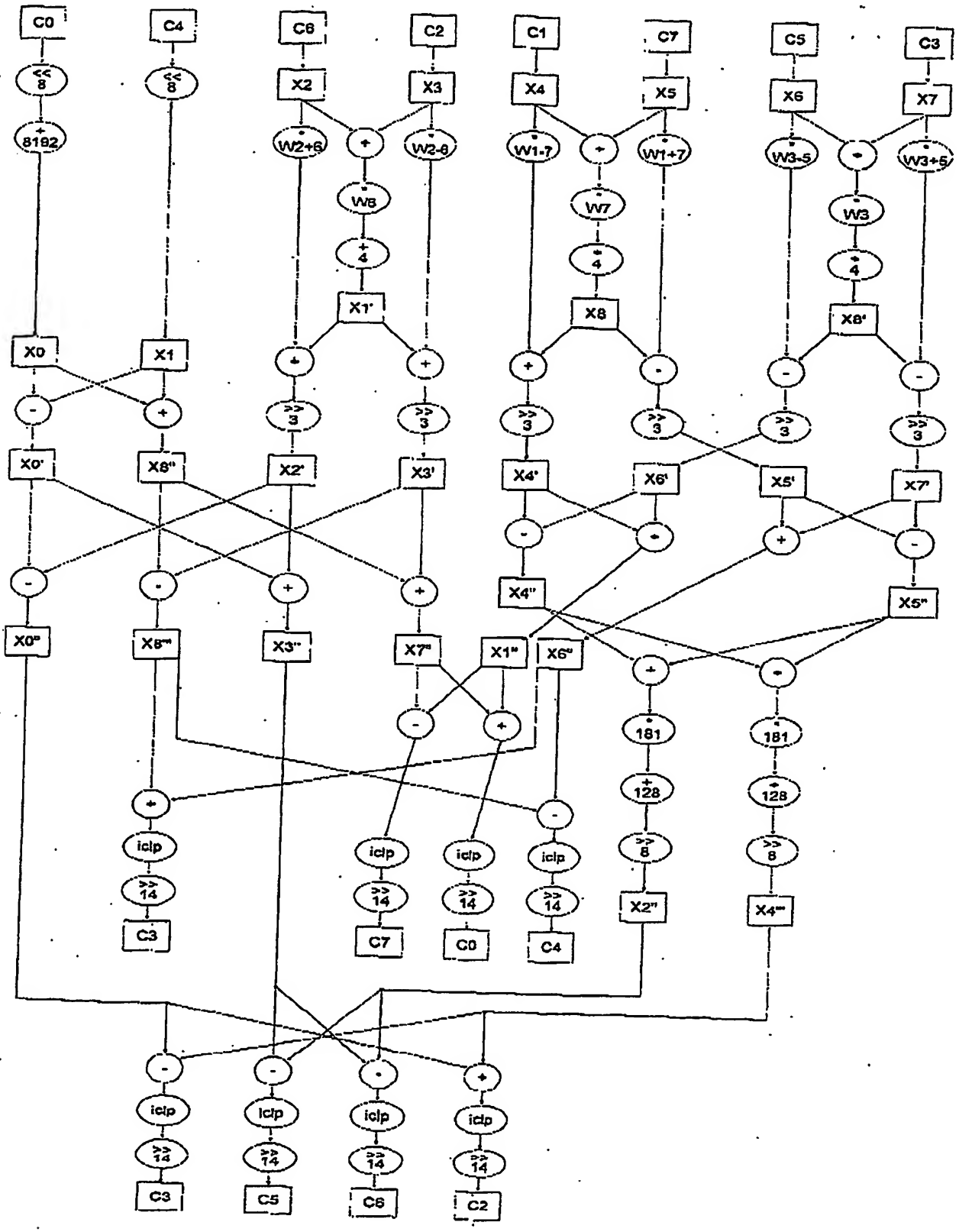
The figure on the next page shows the data flow graph for the IDCTCOLUMN_CONFIG. A heuristic has to be applied to the graph to estimate the resource needs on the XPP. In our example the heuristic produces the following results:

	ADD,SUB	MUL	<< X, >> X	Saturate(x,n)
Ops needed	35	11	18	8

	ALUs	FREGs	BREGs
Res. left	19	80	45
Res. avail.	64	80	80

Fortunately the data flow graph fits into an XPP64 and we can proceed without *loop dissevering* [7] (splitting the loop body into suitable chunks) for this example.

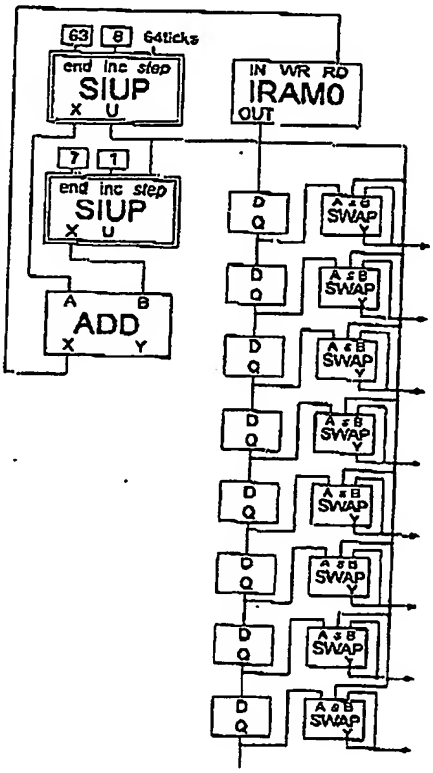
[7] XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture, J. M. P. Cardoso and Markus Weinhardt


C0
C4
C6
C2
C1
C7
C5
C3
<< 8
8192
X2
X3
X4
X5
X6
X7
W2+6
W2-6
W1-7
W1+7
W3-5
W3+5
W8
W7
W3
+ 4
X1'
X8
X8'
X0
X1
X0'
X8''
X2'
X3'
X4'
X6'
X5'
X7'
>> 3
X0''
X8'''
X3''
X7''
X1''
X6''
X4''
X5''
iclp
>> 14
>> 8
181
+ 128
C3
C7
C0
C4
X2''
X4'''
C5
C6
C2

Address Generation:

To fully synthesize the loop body we have to face the problem of address generation for accessing the data.

For IDCTCOLUMN_CONFIG we have to select the n^{th} element of every row which means an address serial of (0,8,16...1,9,17... 7,15,23...). We use two counter macros for address generation as shown opposite. The upper counter increments by eight and the lower by one. The IRAM output is passed to the data flow graph of IDCTCOLUMN. If all (eight) row elements of a column are available SWAP is switched through to the data flow graph input and the calculation for a new column begins.



For the IDCTROW_CONFIG the address generation is very simple as the IRAM already contains the block in the appropriate order (row after row as it has to be accessed). Again by using SIUP(stepped iterative up)-counter macros as described in the XPP tutorial it is possible to map linear address expressions to NML-code in a generic way. As IDCTROW_CONFIG accesses a two-dimensional array we need two SIUP-counters in the corresponding NML code. The column-elements have to be accessed row after row so the upper counters increment is one and the lower counters increment is eight. However, the NML code for this access pattern (0...,5,6,7,8,9,...63) can be reduced to one single counter (or to FIFO-mode IRAM access).

Address generation for write access is implemented in the same manner. The resources have to be updated to take this additional code into account. It takes 2*(8+8+2*1) FREGs and 2*(2+1) more BREGs in the worst case which is still available on the XPP.
If IRAM use is not critical it is also possible to distribute the data on several IRAMs to improve the memory throughput into the XPP-array. This task has to be done by the RISC-core or by a more sophisticated XPP-cache controller.

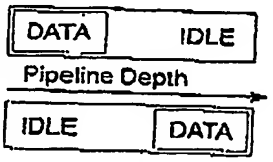
Further Enhancing XPP Utilization

As mentioned at the beginning idct is called for all data blocks of a video image (loop in transform.c). This circumstance allows us to further improve the XPP utilization.

When we look at the data flow graph of idctcol in detail we see that it forms a very deep pipeline. If we bring back to our mind that the IDCTROW_CONFIG runs only eight times on the XPP which meant that only 64 (8 times 8 elements of a column) elements are processed through this pipeline and that we have to wait then until all data left the pipeline before we can change the XPP configuration to the IDCTCOLUMN_CONFIG configuration to go on with column processing then it gets obvious that something is suboptimal in our example.

Problem (Pipeline Depth)

The pipeline is just too deep for processing only eight times eight rows. Filling and flushing a deep pipeline is expensive if only little data is processed with it. First the units at the end of the pipeline are idle and then the units at the begin are unused.



Solution (Loop Tiling)

It is profitable to use loop interchange for moving the dependencies between row and column processing to an outer level of the loop nest. The loop that calls the idct-function (in transform.c) on several blocks of the image is has no loop interchange preventing dependencies. Therefore this loop can be moved inside the loops of column and row processing.

```
// transform.c

..

for (n=0; n<block_count; n++) {
    idct(blocks[k*block_count+n]); // block_count is 6 or 8 or 12
}

..

// idct.c

/* two dimensional inverse discrete cosine transform */
void idct(block)
short *block;
{
  int i;

  for (i=0; i<8; i++)
    idctrow(block+8*i);

  for (i=0; i<8; i++)
    idctcol(block+i);
}
```

loop interchange

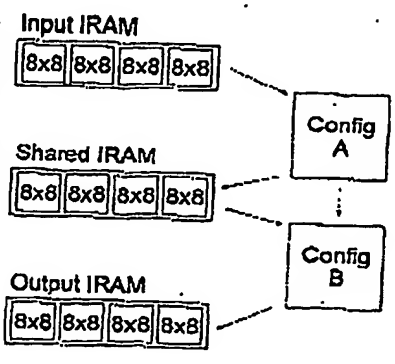
Now the processing of rows and columns can be applied on more data (by applying loop tiling) and therefore filling and flushing the pipeline can be neglected.

Constraints (Cache Sensitive Loop Tiling)

The caching hierarchy has to be taken into account when we define the number of blocks that will be processed by the IDCTROW_CONFIG. Remember, we need the same blocks in the subsequent IDCTCOLUMN_CONFIG configuration! We have to take care that all blocks that are processed during IDCTROW_CONFIG fit into the cache. Loop tiling has to be applied with respect to the cache size so that the processed data fits into the cache.

IRAM reuse between different configurations

This example implies another bandwidth optimization that is just a more consequent version of loop tiling. Instead of transferring data from row processing to column processing via the memory hierarchy (cache sensitive loop tiling takes care that only the cache memory is accessed) we can completely bypass the memory interface by using the output IRAM of Config A as input IRAM of Config B.



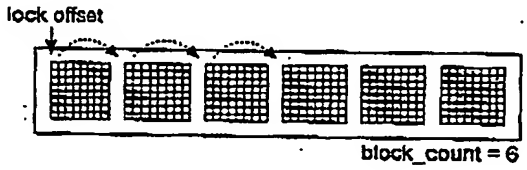
Putting all together

If we apply cache sensitive loop tiling, IRAM reuse and function inlining we can further optimize our example:

Finally the idct-function gets completely inlined in transform.c. If block_count is e.g. 6 and we assume that 64*6 words do not exceed the cache size then we can transform the example to:

```
// transform.c

..

block = blocks[k*6];
XPPPreload(IDCTROW_CONFIG);
XPPPreload(0,block,64*6);       // IRAM0 gets 64 words from 6 blocks
XPPPreloadClean(1,block,64*6); // erase IRAM1 and assign to the 6 blocks
XPPExecute(IDCTROW_CONFIG,IRAM(0),IRAM(1));

XPPPreload(IDCOLUMN_CONFIG);
XPPPreload(1,block,64*6);       // redundant -> will be eliminated
XPPExecute(IDCOLUMN_CONFIG,IRAM(1),IRAM(2));

..
```

The address generation in IDCTROW_CONFIG and IDCOLUMN_CONFIG has to be modified for reflecting the different data block size - caused by loop tiling - that has to be processed. This can be implemented by an additional SUIP counter that generates the block offsets inside the tiles.



The table contains architectural parameters for IDCTROW_CONFIG and IDCOLUMN_CONFIG of the final result. It relies on a cache that is able to store block_count blocks. As two configurations are executed in this example the configuration cycles have to be taken twice and therefore the total configuration cycles are 2 x (block_count x 64 + (12 + 2 x 8) x 2).

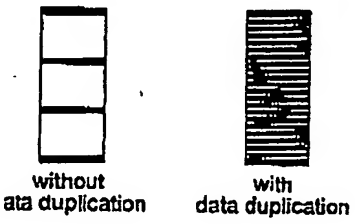
Parameter	Value
Vector length	8 words
Reused data set size	block_count x 64 words
I/O IRAMs	3 (one shared)
ALU	45 FUs
BREG	41 FUs
FREG	36 FUs
Data flow graph width	8
Data flow graph height	12
Configuration cycles	block_count x 64 + (12 + 2*8) x 2

Performance Considerations

In this example it is possible to exploit high data locality which means that many operations are performed on a limited memory range. The performance of the proposed XPP solution is compared to a hypothetical superscalar RISC-architecture. We assume an issue width of two which means that the RISC executes on average two operations in parallel.

	Ops for Row/Column	Est. RISC cycles		
LD/ST	16	2	32	
ADRCOMP	16	1	16	
ADD/SUB	35	1	35	
MULT	11	2	22	
SHIFT	18	1	18	
SAT	8	4	32	
	Issue Width	2	155	
	Cyc/Row(Col)		78	
Proc. Rows	8	620		
Proc. Cols	8	620		
	RISC Cyc/Blk	1240		
	XPP Cyc/Blk	128		with data duplication+reordering 24
	Speedup	10		with data duplication+reordering 52

Even though speedup is reasonable it gets obvious that fetching the input data from a single IRAM (which means that we have to feed the eight inputs in eight cycles before processing is started) reduces the potential speedup significantly. With other words we have a pipeline that is able to process eight input values per cycle but we are loading the pipeline only every eighth cycle. This causes that only every eighth pipeline stage is filled. The figure below illustrates this:



Full utilization can be achieved only by loading the eight input values of the pipeline in one cycle. A simple solution to improve the memory throughput to the pipeline is data duplication as described in the hardware section.

Instead of loading the six 8x8 blocks to a single IRAM we use the **XPPPreloadMultiple** command to load the eight IRAMs with the same contents:

```
XPPPreload(0,block,64*6);        // IRAM0 gets 64 words from 6 blocks
```

is changed to:

```
XPPPreloadMultiple(0xFF, block, 64x6) // load IRAM0 up to IRAM7 with blocks
```

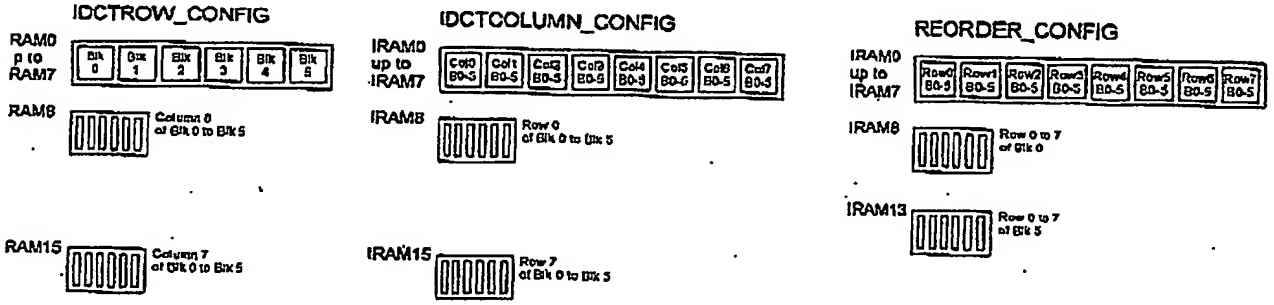
Now the pipeline gets fully utilized and we also have to store eight results per cycle. This can be achieved by writing every output value to another IRAM which additionally takes eight more IRAMs (using data duplication in this example needs all 16 IRAMs of the XPP64). For storing the data that is generated with IDCTROW_CONFIG we have to change:

```
XPPPreloadClean(1,block,64*6); // erase IRAM1 and assign to the 6 blocks
```

to:

```
tmpsize = 64*6/8;
XPPPreloadClean( 8, block+0*tmpsize, tmpsize); // IRAM8  for interm. Rslt 1
XPPPreloadClean( 9, block+1*tmpsize, tmpsize); // IRAM9  for interm. Rslt 1
XPPPreloadClean(10, block+2*tmpsize, tmpsize); // IRAM10 for interm. Rslt 1
XPPPreloadClean(11, block+3*tmpsize, tmpsize); // IRAM11 for interm. Rslt 1
XPPPreloadClean(12, block+4*tmpsize, tmpsize); // IRAM12 for interm. Rslt 1
XPPPreloadClean(13, block+5*tmpsize, tmpsize); // IRAM13 for interm. Rslt 1
XPPPreloadClean(14, block+6*tmpsize, tmpsize); // IRAM14 for interm. Rslt 1
XPPPreloadClean(15, block+7*tmpsize, tmpsize); // IRAM15 for interm. Rslt 1
```

This causes different data layouts for the intermediate results. We need an additional configuration (REORDER_CONFIG) to restore the original data layout.



Again address generation has to be modified to fetch eight input values per cycle. This on the one hand requires seven additional adders, but on the other hand avoids swaps and latches for keeping the data eight cycles.

Data duplication and data reordering finally transforms the example code to:

```
// transform.c

..

block = blocks[k*6];
XPPPreload(IDCTROW_CONFIG);
XPPPreloadMultiple(0xFF, block, 64x6) // load IRAM0 up to IRAM7 with blocks
tmpsize = 64*6/8; // result gets seperated into 8 IRAMs
XPPPreloadClean( 8, block+0*tmpsize, tmpsize); // IRAM8  for interm. Rslt 1
XPPPreloadClean( 9, block+1*tmpsize, tmpsize); // IRAM9  for interm. Rslt 1
XPPPreloadClean(10, block+2*tmpsize, tmpsize); // IRAM10 for interm. Rslt 1
XPPPreloadClean(11, block+3*tmpsize, tmpsize); // IRAM11 for interm. Rslt 1
XPPPreloadClean(12, block+4*tmpsize, tmpsize); // IRAM12 for interm. Rslt 1
XPPPreloadClean(13, block+5*tmpsize, tmpsize); // IRAM13 for interm. Rslt 1
XPPPreloadClean(14, block+6*tmpsize, tmpsize); // IRAM14 for interm. Rslt 1
XPPPreloadClean(15, block+7*tmpsize, tmpsize); // IRAM15 for interm. Rslt 1
XPPExecute(IDCTROW_CONFIG, IRAM(0-7), IRAM(8-15));

XPPPreload(IDCOLUMN_CONFIG);
XPPPreloadMultiple(0xFF, block, 64x6) // ld IRAM0-IRAM7 with interm. Rslt 1
XPPPreloadClean( 8, block+0*tmpsize, tmpsize); // IRAM8  for interm. Rslt 2
XPPPreloadClean( 9, block+1*tmpsize, tmpsize); // IRAM9  for interm. Rslt 2
XPPPreloadClean(10, block+2*tmpsize, tmpsize); // IRAM10 for interm. Rslt 2
```

```
XPPPreloadClean(11, block+3*tmpsize, tmpsize); // IRAM11 for interm. Rslt 2
XPPPreloadClean(12, block+4*tmpsize, tmpsize); // IRAM12 for interm. Rslt 2
XPPPreloadClean(13, block+5*tmpsize, tmpsize); // IRAM13 for interm. Rslt 2
XPPPreloadClean(14, block+6*tmpsize, tmpsize); // IRAM14 for interm. Rslt 2
XPPPreloadClean(15, block+7*tmpsize, tmpsize); // IRAM15 for interm. Rslt 2
XPPExecute(IDCOLUMN_CONFIG,IRAM(0-7),IRAM(8-15));

XPPPreload(REORDER_CONFIG);
XPPPreloadMultiple(0xFF, block, 64x6) // ld IRAM0-IRAM7 with interm. Rslt 2
rsltsize = 64; // 64*6/6;
XPPPreloadClean( 8, block+0*rsltsize, rsltsize); // IRAM8  for final Rslt
XPPPreloadClean( 9, block+1*rsltsize, rsltsize); // IRAM9  for final Rslt
XPPPreloadClean(10, block+2*rsltsize, rsltsize); // IRAM10 for final Rslt
XPPPreloadClean(11, block+3*rsltsize, rsltsize); // IRAM11 for final Rslt
XPPPreloadClean(12, block+4*rsltsize, rsltsize); // IRAM12 for final Rslt
XPPPreloadClean(13, block+4*rsltsize, rsltsize); // IRAM13 for final Rslt
XPPExecute(IDCOLUMN_CONFIG,IRAM(0-7),IRAM(8-13));
```

5.6 Wavelet

5.6.1 Original Code

```
void forward_wavelet()
{
  int i,nt, *dmid;
  int *sp, *dp, d_tmp0, d_tmp1, d_tmpi, s_tmp0, s_tmp1;
  int mid, ii;
  int *x;
  int s[256],d[256];

  for (nt=COL;nt>=BLOCK_SIZE;nt>>=1) {
    for (i=0;i<nt*COL/*tmp_nt*/;i+=COL) {

      x = &int_data[i];
      mid=(nt>>1)-1;

      s[0] = x[0];
      d[0] = x[ROW];
      s[1] = x[2];
      s[mid] = x[2*mid];
      d[mid] = x[2*mid+ROW];

      d[0]=(d[0]<<1)-s[0]-s[1];
      s[0]=s[0]+(d[0]>>2);

      d_tmp0 = d[0];
      s_tmp0 = s[1];

      for(ii=1; ii<mid; ii++){
        s_tmp1 = x[2*ii+2];
        d_tmp1 =((x[2*ii+ROW])<<1) - s_tmp0 - s_tmp1;
        d[ii] = d_tmp1;
        s[ii]= s_tmp0+((d_tmp0+d_tmp1)>>3);
        d_tmp0 = d_tmp1;
        s_tmp0 = s_tmp1;
      }
      d[mid]=(d[mid]-s[mid])<<1;
      s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

      for(ii=0; ii<=mid; ii++) {
        x[ii]=s[ii];
        x[ii+mid+1]=d[ii];
      }
    }

    for (i=0;i<nt;i++) {

      x = &int_data[i];
      mid=(nt>>1)-1;

      s[0] = x[0];
      d[0] = x[COL];
      s[1] = x[COL<<1];
      s[mid] = x[(COL<<1)*mid];
      d[mid] = x[(COL<<1)*mid +COL];
```

```
        d[0]=(d[0]<<1)-s[0]-s[1];
        s[0]=s[0]+(d[0]>>2);

        d_tmp0 = d[0];
        s_tmp0 = s[1];
        for(ii=1; ii<mid; ii++) {
          s_tmp1 = x[2*COL*(ii+1)];
          d_tmp1 =(x[2*COL*ii+COL]<<1) - s_tmp0 - s_tmp1;
          d[ii] = d_tmp1;
          s[ii]= s_tmp0+((d_tmp0+d_tmp1)>>3);
          d_tmp0 = d_tmp1;
          s_tmp0 = s_tmp1;
        }

        d[mid]=(d[mid]<<1) -(s[mid]<<1);
        s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

        for(ii=0; ii<=mid; ii++) {
          x[ii*COL]=s[ii];
          x[(ii+mid+1)*COL]=d[ii];
        }
      }
    }

}
```

5.6.2 Optimizing the Whole Loop Nest

After pre-processing and application of copy propagation over *s_tmp1*, *d_tmp1*, we obtain the following loop nest

```
void forward_wavelet()
{
  int i,nt, *dmid;
  int *sp, *dp, d_tmp0, d_tmp1, d_tmpi, s_tmp0, s_tmp1;
  int mid, ii;
  int *x;
  int s[256],d[256];

  for (nt=64;nt>= 16;nt>>=1) {
    for (i=0;i<nt*64;i+=64) {

      x = &int_data[i];
      mid=(nt>>1)-1;

      s[0] = x[0];
      d[0] = x[1];
      s[1] = x[2];
      s[mid] = x[2*mid];
      d[mid] = x[2*mid+1];

      d[0]=(d[0]<<1)-s[0]-s[1];
      s[0]=s[0]+(d[0]>>2);

      d_tmp0 = d[0];
      s_tmp0 = s[1];
```

```
      for(ii=1; ii<mid; ii++){
        d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
        s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
        d_tmp0 = d[ii];
        s_tmp0 = s[ii];
      }
      d[mid]=(d[mid]-s[mid])<<1;
      s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

      for(ii=0; ii<=mid; ii++) {
        x[ii]=s[ii];
        x[ii+mid+1]=d[ii];
      }
    }

    for (i=0;i<nt;i++) {
      x = &int_data[i];
      mid=(nt>>1)-1;

      s[0] = x[0];
      d[0] = x[64];
      s[1] = x[128];
      s[mid] = x[128*mid];
      d[mid] = x[128*mid +64];

      d[0]=(d[0]<<1)-s[0]-s[1];
      s[0]=s[0]+(d[0]>>2);

      d_tmp0 = d[0];
      s_tmp0 = s[1];

      for(ii=1; ii<mid; ii++) {
        d[ii] =(x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
        s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
        d_tmp0 = d[ii];
        s_tmp0 = s[ii];
      }

      d[mid]=(d[mid]<<1) -(s[mid]<<1);
      s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

      for(ii=0; ii<=mid; ii++) {
        x[ii*64]=s[ii];
        x[(ii+mid+1)*64]=d[ii];
      }
    }
  }

}
```

Then we have 4 tables, one for each innermost loop. The tables for the first and the third loops are identical, as are the tables for the second and the fourth loop. We have the following two tables.

Parameter	Value
Vector length	mid-2
Reused data set size	-
I/O IRAMs	6
ALU	6
BREG	0
FREG	2
Data flow graph width	2
Data flow graph height	6
Configuration cycles	6+(mid-2)

Parameter	Value
Vector length	mid
Reused data set size	-
I/O IRAMs	6
ALU	0
BREG	0
FREG	0
Data flow graph width	2
Data flow graph height	1
Configuration cycles	mid

The two inner loops do not have the same iteration range and could be candidate for loop fusion, therefore the first and last iterations of the second loop are peeled off. The surrounding code between the 2 loops can be moved after the second loop, then we obtain the following code for the loop nest.

```
for (nt=64;nt>= 16;nt>>=1) {
  for (i=0;i<nt*64;i+=64) {
    x = &int_data[i];
    mid=(nt>>1)-1;

    s[0] = x[0];
    d[0] = x[1];
    s[1] = x[2];

    s[mid] = x[2*mid];
    d[mid] = x[2*mid+1];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];
```

```
    for(ii=1; ii<mid; ii++){
      d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
    }

    for(ii=1; ii<mid; ii++) {
      x[ii]=s[ii];
      x[ii+mid+1]=d[ii];
    }

    d[mid]=(d[mid]-s[mid])<<1;
    s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

    x[0]=s[0];
    x[mid+1]=d[0];
    x[mid]=s[mid];
    x[2*mid+1]= d[mid];
  }

  for (i=0;i<nt;i++) {
    x = &int_data[i];
    mid=(nt>>1)-1;

    s[0] = x[0];
    d[0] = x[64];
    s[1] = x[128];
    s[mid] = x[128*mid];
    d[mid] = x[128*mid +64];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];
    for(ii=1; ii<mid; ii++) {
      d[ii] =(x[128 *ii+64]<<1) - s_tmp0 - x[128 *(ii+1)];
      s[ii]= s_tmp0+((d_tmp0+d_tmp1)>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
    }

    for(ii=1; ii<mid; ii++) {
      x[ii*64]=s[ii];
      x[(ii+mid+1)*64]=d[ii];
    }
    d[mid]=(d[mid]<<1) -(s[mid]<<1);
    s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

    x[0]=s[0];
    x[(mid+1)*64]=d[0];
    x[mid*64]=s[mid];
    x[(2*mid+1)*64]=d[mid];
  }
}
```

After loop peeling the only change on the parameters is the vector length. The tables become:

Parameter	Value
Vector length	mid-2
Reused data set size	-
I/O IRAMs	6
ALU	2
BREG	0
FREG	2
Data flow graph width	2
Data flow graph height	6
Configuration cycles	6+(mid-2)

Parameter	Value
Vector length	mid-2
Reused data set size	-
I/O IRAMs	6
ALU	0
BREG	0
FREG	0
Data flow graph width	2
Data flow graph height	1
Configuration cycles	mid-2

The fusion of the inner loops is legal as there would be no loop-carried dependences between the instructions formerly in the second loop and the instructions formerly in the first loop. We obtain the following loop nest.

```
for (nt=64;nt>= 16;nt>>=1) {
  for (i=0;i<nt*64 /*tmp_nt*/;i+=64) {

    x = &int_data[i];
    mid=(nt>>1)-1;

    s[0] = x[0];
    d[0] = x[1];
    s[1] = x[2];
    s[mid] = x[2*mid];
    d[mid] = x[2*mid+1];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];
```

```
    for(ii=1; ii<mid; ii++){
      d[ii] = ((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
      s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii] = s[ii];
      x[ii+mid+1] = d[ii];
    }
    d[mid]=(d[mid]-s[mid])<<1;
    s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

    x[0]=s[0];
    x[mid+1]=d[0];
    x[mid]=s[mid];
    x[2*mid+1]= d[mid];
  }

  for (i=0;i<nt;i++) {

    x = &int_data[i];
    mid=(nt>>1)-1;

    s[0] = x[0];
    d[0] = x[64];
    s[1] = x[128];
    s[mid] = x[128*mid];
    d[mid] = x[128*mid +64];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];

    for(ii=1; ii<mid; ii++) {
      d[ii] =(x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii*64]=s[ii];
      x[(ii+mid+1)*64]=d[ii];
    }
    d[mid]=(d[mid]<<1) -(s[mid]<<1);
    s[mid]=s[mid]+((d[mid-1]+d[mid])>>3);

    x[0]=s[0];
    x[(mid+1)*64]=d[0];
    x[mid*64]=s[mid];
    x[(2*mid+1)*64]=d[mid];
  }
}
```

After loop fusion, we only have two loops, that have the same parameter table.

Parameter	Value
Vector length	mid-2
Reused data set size	-
I/O IRAMs	8
ALU	6
BREG	0
FREG	2
Data flow graph width	2
Data flow graph height	6
Configuration cycles	6+(mid-2)

When performing value range analysis, the compiler finds that *nt* ranges takes the values 64, 32 and 16. The upper bound of the inner loops is *mid*, which depends on the value of *nt*. The analysis finds then that *mid* can take the values: 31, 15 and 7. Loops with constant loop bounds can be handled more efficiently on the PACT XPP. This means that the inner loops can be better optimized if *mid* is replaced by a constant value. This will happen when the outer loop is unrolled. This way we will obtain a bigger code, but with 3 instances of the loop nest, each being candidate for a configuration. This can be seen as a kind of temporal partitioning. Thus the outer loop is completely unrolled giving six new loop nests.

```
for (i=0;i<4096;i+=64) { /* nt=64 */

  x = &int_data[i];
  mid=31;

  s[0] = x[0];
  d[0] = x[1];
  s[1] = x[2];
  s[31] = x[61];
  d[31] = x[63];

  d[0]=(d[0]<<1)-s[0]-s[1];
 s[0]=s[0]+(d[0]>>2);

  d_tmp0 = d[0];
  s_tmp0 = s[1];

  for(ii=1; ii<31; ii++){
    d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
    s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
    d_tmp0 = d[ii];
    s_tmp0 = s[ii];
    x[ii]=s[ii];
    x[ii+32]=d[ii];
  }
  d[31]=(d[31]-s[31])<<1;
  s[31]=s[31]+((d[30]+d[31])>>3);
```

```
    x[0]=s[0];
    x[32]=d[0];
    x[31]=s[31];
    x[63]=d[31];
  }

  for (i=0;i<64;i++) {

    x = &int_data[i];
    mid=31;

    s[0] = x[0];
    d[0] = x[64];
    s[1] = x[128];
    s[31] = x[3968];
    d[31] = x[4032];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];

    for(ii=1; ii<31; ii++) {
      d[ii] =(x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii*64]=s[ii];
      x[(ii+32)*64]=d[ii];
    }
    d[31]=(d[31]<<1) -(s[31]<<1);
    s[31]=s[31]+((d[30]+d[31])>>3);

    x[0]=s[0];
    x[2048]=d[0];
    x[1984]=s[31];
    x[4032]=d[31];
  }

  for (i=0;i<2048;i+=64) { /* nt = 32 */

    x = &int_data[i];
    mid=15;

    s[0] = x[0];
    d[0] = x[1];
    s[1] = x[2];
    s[15] = x[30];
    d[15] = x[31];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];
```

```
    for(ii=1; ii<15; ii++){
      d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii]=s[ii];
      x[ii+16]=d[ii];
    }
    d[15]=(d[15]-s[15])<<1;
    s[15]=s[15]+((d[14]+d[15])>>3);

    x[0]=s[0];
    x[16]=d[0];
    x[15]=s[15];
    x[31]= d[15];
  }

  for (i=0;i<32;i++) {

    x = &int_data[i];
    mid=15;

    s[0] = x[0];
    d[0] = x[64];
    s[1] = x[128];
    s[15] = x[1920];
    d[15] = x[1984];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];

    for(ii=1; ii<15; ii++) {
      d[ii] =(x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii*64]=s[ii];
      x[(ii+16)*64]=d[ii];
    }
    d[15]=(d[15]<<1) -(s[15]<<1);
    s[15]=s[15]+((d[14]+d[15])>>3);

    x[0]=s[0];
    x[1024]=d[0];
    x[960]=s[15];
    x[1984]=d[15];
  }

  for (i=0;i<1024;i+=64) { /* nt = 16 */

    x = &int_data[i];
    mid=7;

    s[0] = x[0];
    d[0] = x[1];
    s[1] = x[2];
    s[7] = x[14];
    d[7] = x[15];
```

```
    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];

    for(ii=1; ii<7; ii++){
      d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii]=s[ii];
      x[ii+8]=d[ii];
    }
    d[7]=(d[7]-s[7])<<1;
    s[7]=s[7]+((d[6]+d[7])>>3);

    x[0]=s[0];
    x[8]=d[0];
    x[7]=s[7];
    x[15]= d[7];
  }

  for (i=0;i<16;i++) {

    x = &int_data[i];
    mid=7;

    s[0] = x[0];
    d[0] = x[64];
    s[1] = x[128];
    s[7] = x[896];
    d[7] = x[960];

    d[0]=(d[0]<<1)-s[0]-s[1];
    s[0]=s[0]+(d[0]>>2);

    d_tmp0 = d[0];
    s_tmp0 = s[1];

    for(ii=1; ii<7; ii++) {
      d[ii] =(x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
      s[ii]= s_tmp0+((d_tmp0 + d[ii])>>3);
      d_tmp0 = d[ii];
      s_tmp0 = s[ii];
      x[ii*64]=s[ii];
      x[(ii+8)*64]=d[ii];
    }
    d[7]=(d[7]<<1) -(s[7]<<1);
    s[7]=s[7]+((d[6]+d[7])>>3);

    x[0]=s[0];
    x[512]=d[0];
    x[448]=s[7];
    x[960]=d[7];
  }
```

In the parameter table, the vector length is the only value that change. We give it for the first two loops. To deduce the table for the other loops, the vector length has to be set to 14 and 6 respectively.

Parameter	Value
Vector length	30
Reused data set size	-
I/O IRAMs	8
ALU	6
BREG	0
FREG	2
Data flow graph width	2
Data flow graph height	6
Configuration cycles	6+30=36

5.6.3 Optimizing the Inner Loops

The efforts are then concentrated on the six inner loops. In fact, if we look at them, they all need 2 input data and output 4 data. 2 more data are needed for the first iteration. Hence we need at most 8 IRAMs for the first iteration and 6 for the others. This means that we can unroll the loops twice, needing 14 IRAMs for one iteration of the new loop bodies. Below we present only the unrolled inner loops for commodity reasons.

First loop:

```
for(ii=1; ii<31; ii=ii+2){
   d[ii] =((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
   s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
   d_tmp0 = d[ii];
   s_tmp0 = s[ii];
   x[ii+1] = s[ii];
   x[ii+33]=d[ii];
   d[ii+1] =((x[2*(ii+1)+1])<<1) - s_tmp0 - x[2*(ii+1)+2];
   s[ii+1] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
   d_tmp0 = d[ii+1];
   s_tmp0 = s[ii+1];
   x[ii+1] = s[ii+1];
   x[ii+33] = d[ii+1];
}
```

Second loop:

```
for(ii=1; ii<31; ii=ii+2) {
   d[ii] = (x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
   s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
   d_tmp0 = d[ii];
   s_tmp0 = s[ii];
   x[ii*64] = s[ii];
   x[(ii+32)*64] = d[ii];
   d[ii+1] =(x[128*(ii+1)+64]<<1) - s_tmp0 - x[128*(ii+2)];
   s[ii+1] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
   d_tmp0 = d[ii+1];
   s_tmp0 = s[ii+1];
   x[(ii+1)*64] = s[ii+1];
   x[(ii+33)*64] = d[ii+1];
}
```

Third loop:

```
for(ii=1; ii<15; ii=ii+2){
    d[ii] = ((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
    s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
    d_tmp0 = d[ii];
    s_tmp0 = s[ii];
    x[ii] = s[ii];
    x[ii+16] = d[ii];
    d[ii+1] = ((x[2*(ii+1)+1])<<1) - s_tmp0 - x[2*(ii+1)+2];
    s[ii+1] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
    d_tmp0 = d[ii+1];
    s_tmp0 = s[ii+1];
    x[ii+1] = s[ii+1];
    x[ii+17] = d[ii+1];
}
```

Fourth loop:

```
for(ii=1; ii<15; ii=ii+2) {
    d[ii] = (x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
    s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
    d_tmp0 = d[ii];
    s_tmp0 = s[ii];
    x[ii*64] = s[ii];
    x[(ii+16)*64] = d[ii];
    d[ii+1] = (x[128*(ii+1)+64]<<1) - s_tmp0 - x[128*(ii+2)];
    s[ii] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
    d_tmp0 = d[ii+1];
    s_tmp0 = s[ii+1];
    x[(ii+1)*64] = s[ii+1];
    x[(ii+17)*64] = d[ii+1];
}
```

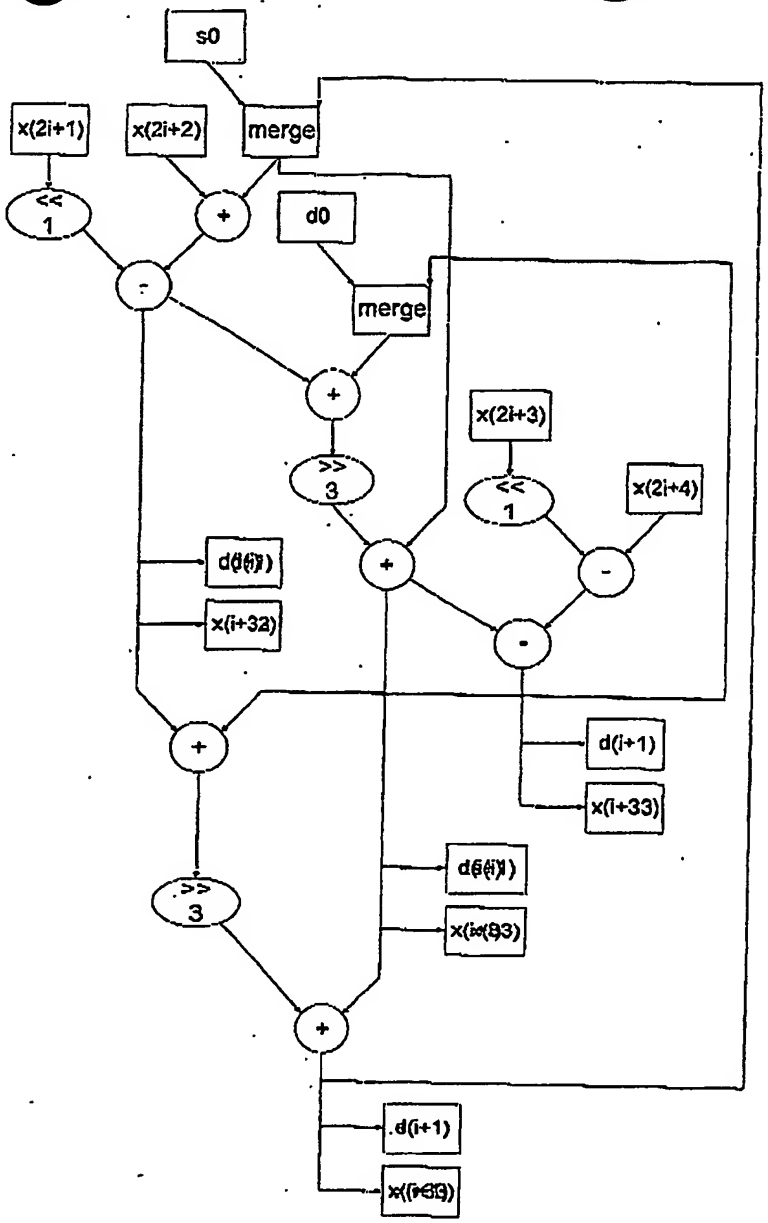
Fifth loop:

```
for(ii=1; ii<7; ii=ii+2){
    d[ii] = ((x[2*ii+1])<<1) - s_tmp0 - x[2*ii+2];
    s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
    d_tmp0 = d[ii];
    s_tmp0 = s[ii];
    x[ii] = s[ii];
    x[ii+8] = d[ii];
    d[ii+1] = ((x[2*(ii+1)+1])<<1) - s_tmp0 - x[2*(ii+1)+2];
    s[ii+1] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
    d_tmp0 = d[ii+1];
    s_tmp0 = s[ii+1];
    x[ii+1] = s[ii+1];
    x[ii+9] = d[ii+1];
}
```

Sixth loop:

```
for(ii=1; ii<7; ii=ii+2) {
    d[ii] = (x[128*ii+64]<<1) - s_tmp0 - x[128*(ii+1)];
    s[ii] = s_tmp0+((d_tmp0 + d[ii])>>3);
    d_tmp0 = d[ii];
    s_tmp0 = s[ii];
    x[ii*64] = s[ii];
    x[(ii+8)*64] = d[ii];
    d[ii+1] = (x[128*(ii+1)+64]<<1) - s_tmp0 - x[128*(ii+2)];
    s[ii] = s_tmp0+((d_tmp0 + d[ii+1])>>3);
    d_tmp0 = d[ii+1];
    s_tmp0 = s[ii+1];
    x[(ii+1)*64] = s[ii+1];
    x[(ii+9)*64] = d[ii+1];
}
```

We obtain the following dataflow graph of these loop bodies after a step of tree balancing has been performed. We represent here only the graph corresponding to the first loop. To obtain the graphs for the other loops, only the input and output data need to be changed.



Each input and output data will occupy an IRAM. *d0* and *s0* will be the only values in their IRAM, enabling then the merge operations to select between *d0*, resp. *s0* at the first iteration and the feedback values for the other iterations. Once the pipeline is filled, 8 values can be output in a cycle, corresponding to 4 values for array *x*. The same configuration is used for all loops; only the data in the IRAMs differ. We give now result tables only for the 2 first loops. The other tables are the same.

For the first two loops we obtain the following table, and the expected speedup with respect to a standard superscalar processor with 2 instructions issued per cycle is 15.3.

Parameter	Value
Vector length	30
Reused data set size	-
I/O IRAMs	14
ALU	12
BREG	0
FREG	2
Data flow graph width	2
Data flow graph height	10
Configuration cycles	10+15=25

Ops	Number
LD/ST (2 cycles)	14
ADDRCOMP (1 cycle)	2
ADD/SUB (1 cycle)	17
MUL (2 cycles)	0
SHIFT (1 cycle)	4
Cycles per iteration	51
Cycles needed for the loop (2-way)	(51*15)/2=383

6 References

[1] Markus Weinhardt and Wayne Luk. Memory Access Optimization for Reconfigurable Systems. *IEE Proceedings Computers and Digital Techniques*, 48(3), May 2001.

[2] Michael Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, 1996.

[3] Hans Zima and Barbara Chapman. Supercompilers for parallel and vector computers. Addison-Wesley, 1991.

[4] David F. Bacon, Susan L. Graham and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):325-420,1994.

[5] Steven Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[6] João M.P. Cardoso and Markus Weinhardt. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, FPL'2002*, volume 2438 of LNCS, pages 864-874, Montpellier, France, 2002.

[7] Markus Weinhardt and Wayne Luk. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234-248, February 2001.

[8] V. Baumgarte, F. May, A. Nückel, M. Vorbach and M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. In *Proceedings of the 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2001)*, Las Vegas, Nevada, 2001.

[9] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171-210,2002.

[10] Sam. Larsen, Emmett Witchel and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 2002 IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, pages 18-29, Charlottesville, Virginia, September, 2002.

[11] Daniela Genius and Sylvain Lelait. A Case for Array Merging in Memory Hierarchies. In *Proceedings of the 9th International Workshop on Compilers for Parallel Computers, CPC'01, Edinburgh, Scotland, June 27th-29th 2001.*

The Invention will now be described further and/or in other details by the following part of the description entitled "A method for Compiling High-level language 137 Programs to a Reconfigurable

1 Introduction

This document describes a method for compiling a subset of a high-level programming language (HLL) like C or FORTRAN, extended by port access functions, to a reconfigurable data-flow processor (RDFP) as described in Section 3. The program is transformed to a configuration of the RDFP.

This method can be used as part of an extended compiler for a hybrid architecture consisting of standard host processor and a reconfigurable data-flow coprocessor. The extended compiler handles a full HLL like standard ANSI C. It maps suitable program parts like inner loops to the coprocessor and the rest of the program to the host processor. It is also possible to map separate program parts to separate configurations. However, these extensions are not subject of this document.

2 Compilation Flow

This section briefly describes the phases of the compilation method.

2.1 Frontend

The compiler uses a standard frontend which translates the input program (e. g. a C program) into an internal format consisting of an abstract syntax tree (AST) and symbol tables. The frontend also performs well-known compiler optimizations as constant propagation, dead code elimination, common subexpression elimination etc. For details, refer to any compiler construction textbook like [1]. The SUIF compiler [2] is an example of a compiler providing such a frontend.

2.2 Control/Dataflow Graph Generation

Next, the program is mapped to a control/dataflow graph (CDFG) consisting of connected RDFP functions. This phase is the main subject of this document and presented in Section 4.

2.3 Configuration Code Generation

Finally, the last phase directly translates the CDFG to configuration code used to program the RDFP. For PACT XPP™ Cores, the configuration code is generated as an NML (Native Mapping Language) file.

3 Configurable Objects and Functionality of a RDFP

This section describes the configurable objects and funcitionality of a RDFP. A possible implementation of the RDFP architecture is a PACT XPP™ Core. Here we only describe the minimum requirements for a RDFP for this compilation method to work. The only data types considered are multi-bit words called *data* and single-bit control signals called *events*. Data and events are always processed as *packets*, cf. Section 3.2. Event packets are called 1-events or 0-events, depending on their bit-value.

3.1 Configurable Objects and Functions

An RDFP consists of an array of configurable objects and a communication network. Each object can be configured to perform certain functions (listed below). It performs the same function repeatedly until the configuration is changed. The array needs not be completely uniform, i. e. not all objects need to be able to perform all functions. E. g., a RAM function can be implemented by a specialized RAM object which cannot perform any other functions. It is also possible to combine several objects to a "macro" to realize certain functions. Several RAM objects can, e. g. , be combined to realize a RAM function with larger storage.

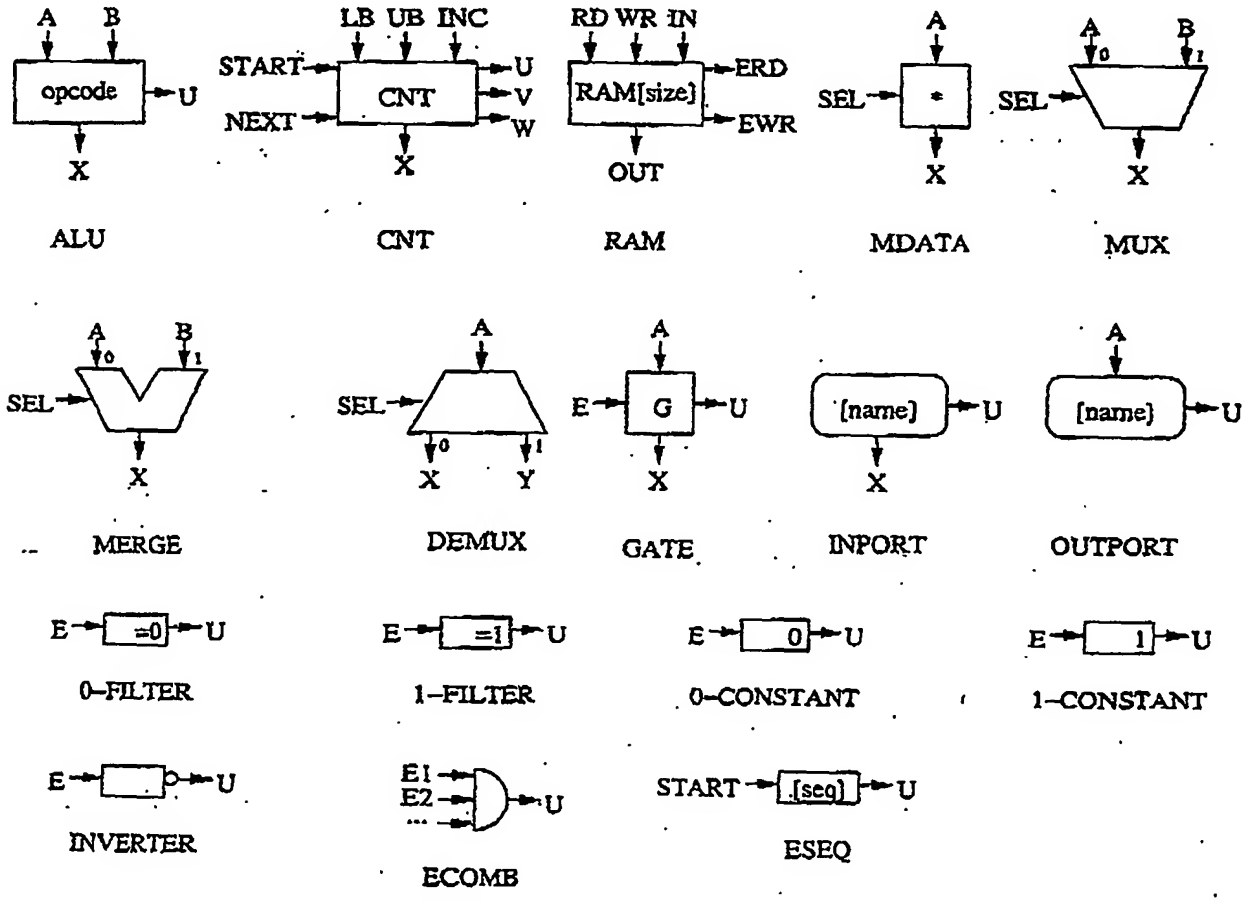


Figure 1: Functions of an RDFP

The following functions for processing data and event packets can be configured into an RDFP. See Fig. 1 for a graphical representation.

- ALU[opcode]: ALUs perform common arithmetical and logical operations on data. ALU functions ("opcodes") must be available for all operations used in the HLL.[1] ALU functions have two data inputs A and B, and one data output X. Comparators have an event output U instead of the data output. They produce a 1-event if the comparison is true, and a 0-event otherwise.

[1]Otherwise programs containing operations which do not have ALU opcodes in the RDFP must be excluded from the supported HLL subset or substituted by "macros" of existing functions.

- CNT: A counter function which has data inputs LB, UB and INC (lower bound, upper bound and increment) and data output X (counter value). A packet at event input START starts the counter, and event input NEXT causes the generation of the next output value (and output events) or causes the counter to terminate if UB is reached. If NEXT is not connected, the counter counts continuously. The output events U, V, and W have the following functionality: For a counter counting N times, N-1 0-events and one 1-event are generated at output U. At output V, N 0-events are generated, and at output W, N 0-events and one 1-event are created. The 1-event at W is only created after the counter has terminated, i.e. a NEXT event packet was received after the last data packet was output.
- RAM[size]: The RAM function stores a fixed number of data words ("size"). It has a data input RD and a data output OUT for reading at address RD. Event output ERD signals completion of the read access. For a write access, data inputs WR and IN (address and value) and data output OUT is used. Event output EWR signals completion of the write access. ERD and EWR always generate 0-events. Note that external RAM can be handled as RAM functions exactly like internal RAM.
- GATE: A GATE synchronizes a data packet at input A back and an event packet at input E. When both inputs have arrived, they are both consumed. The data packet is copied to output X, and the event packet to output U.
- MUX: A MUX function has 2 data inputs A and B, an event input SEL, and a data output X. If SEL receives a 0-event, input A is copied to output X and input B discarded. For a 1-event, B is copied and A discarded.
- MERGE: A MERGE function has 2 data inputs A and B, an event input SEL, and a data output X. If SEL receives a 0-event, input A is copied to output X, but input B is *not* discarded. The packet is left at the input B instead. For a 1-event, B is copied and A left at the input.
- DEMUX: A DEMUX function has one data input A, an event input SEL, and two data outputs X and Y. If SEL receives a 0-event, input A is copied to output X, and no packet is created at output Y. For a 1-event, A is copied to Y, and no packet is created at output X.
- MDATA: A MDATA function multiplicates data packets. It has a data input A, an event input SEL, and a data output X. If SEL receives a 1-event, a data packet at A is consumed and copied to output X. For all subsequent 0-event at SEL, a copy of the input data packet is produced at the output without consuming new packets at A. Only if another 1-event arrives at SEL, the next data packet at A is consumed and copied.[2]
- INPORT[name]: Receives data packets from outside the RDFP through input port "name" and copies them to data output X. If a packet was received, a 0-event is produced at event output U, too. (Note that this function can only be configured at special objects connected to external busses.)
- OUTPORT[name]: Sends data packets received at data input A to the outside of the RDFP through output port "name". If a packet was sent, a 0-event is produced at event output U, too. (Note that this function can only be configured at special objects connected to external busses.)

Additionally, the following functions manipulate only event packets:

[2]Note that this can be implemented by a MERGE with special properties on XPP™.

- 0-FILTER, 1-FILTER: A FILTER has an input E and an output U. A 0-FILTER copies a 0-event from E to U, but 1-EVENTs at E are discarded. A 1-FILTER copies 1-events and discards 0-events.
- INVERTER: Copies all events from input E to output U but inverts its value.
- 0-CONSTANT, 1-CONSTANT: 0-CONSTANT copies all events from input E to output U, but changes them all to value 0. 1-CONSTANT changes all to value 1.
- ECOMB: Combines two or more inputs E1, E2, E3..., producing a packet at output U. The output is a 1-event if and only if one or more of the input packets are 1-events (logical *or*). A packet must be available at all inputs before an ouput packet is produced.[3]
- ESEQ[seq]: An ESEQ generates a sequence "seq" of events, e.g. "0001", at its output U. If it has an input START, one entire sequence is generated for each event packet arriving at U. The sequence is only repeated if the next event arrives at U. However, if START is not connected, ESEQ constantly repeats the sequence.

Note that ALU, MUX, DEMUX, GATE and ECOMB functions behave like their equivalents in classical dataflow machines [3, 4].

3.2 Packet-based Communication Network

The communication network of an RDFP can connect an outputs of one object (i.e. its respective function) to the input(s) of one or several other objects. This is usually achieved by busses and switches. By placing the functions properly on the objects, many functions can be connected arbitrarily up to a limit imposed by the device size. As mentioned above, all values are communicated as packets. A separate communication network exists for data and event packets. The packets synchronize the functions as in a dataflow machine with acknowledge [3]. I. e., the function only executes when all input packets are available (apart from the non-strict exceptions as described above). The function also stalls if the last output packet has not been consumed. Therefore a data-flow graph mapped to an RDFP self-synchronizes its execution without the need for external control. Only if two or more function outputs (data or event) are connected to the same function input ("N to 1 connection"), the self-synchronization is disabled.[4] The user has to ensure that only one packet arrives at a time in a correct CDFG. Otherwise a packet might get lost, and the value resulting from combining two or more packets is undefined. However, a function output can be connected to many function inputs ("1 to N connection") without problems.

There are some special cases:

- A function input can be *preloaded* with a distinct value during configuration. This packet is consumed like a normal packet coming from another object.
- A function input can be defined as *constant*. In this case, the packet at the input is reproduced repeatedly for each function execution.

[3]Note that this function is implemented by the EAND operator on the XPP™.

[4]Note that on XPP™ Cores, a "N to 1 connection" for events is realized by the EOR function, and for data by just assigning several outputs to an input.

An RDFP requires register delays in the dataflow. Otherwise very long combinational delays and asynchronous feedback is possible. We assume that delays are inserted at the inputs of some functions (like for most ALUs) and in some routing segments of the communication network. Note that registers change the timing, but not the functionality of a correct CDFG.

4 Configuration Generation

4.1 Language Definition

The following HLL features are not supported by the method described here:

- pointer operations
- library calls, operating system calls (including standard I/O functions)
- recursive function calls (Note that non-recursive function calls can be eliminated by function inlining and therefore are not considered here.)
- All scalar data types are converted to type integer. Integer values are equivalent to *data* packets in the RDFP. Arrays (possibly multi-dimensional) are the only composite data types considered.

The following additional features are supported:

INPORTS and OUTPORTS can be accessed by the HLL functions *getstream(name, value)* and *putstream(name, value)* respectively.

4.2 Mapping of High-Level Language Constructs

This method converts a HLL program to a CDFG consisting of the RDFP functions defined in Section 3.1. Before the processing starts, all HLL program arrays are mapped to RDFP RAM functions. An array x is mapped to RAM RAM(x). If several arrays are mapped to the same RAM, an offset is assigned, too. The RAMs are added to an initially empty CDFG. There must be enough RAMs of sufficient size for all program arrays.

The CDFG is generated by a traversal of the AST of the HLL program. It processes the program statement by statement and descends into the loops and conditional statements as appropriate. The following two pieces of information are updated at every program point[5] during the traversal:

- START points to an event output of a RDFP function. This output delivers a 0-event whenever the program execution reaches this program point. At the beginning, a 0-CONSTANT preloaded with an event input is added to the CDFG. (It delivers a 0-event immediately after configuration.) START initially points to its output. This event is used to start the overall program execution. The $START_{new}$ signal generated after a program part has finished executing is used as new START signal for the following program parts, or it signals termination of the entire program. The START

[5] In a program, *program points* are between two statements or before the beginning or after the end of a program component like a loop or a conditional statement.

events guarantee that the execution order of the original program is maintained wherever the data dependencies alone are not sufficient. This scheduling scheme is similar to a *one-hot controller* for digital hardware.

- VARLIST is a list of {*variable, function-output*} pairs. The pairs map integer variables or array elements to a CDFG function's output. The first pair for a variable in VARLIST contains the output of the function which produces the value of this variable valid at the current program point. New pairs are always added to the front of VARLIST. The expression VARDEF(var) refers to the *function-output* of the first pair with *variable* var in VARLIST.[6]

The following subsections systematically list all HLL program components and describe how they are processed, thereby altering the CDFG, START and VARLIST.

4.2.1 Integer Expressions and Assignments

Straight-line code without array accesses can be directly mapped to a data-flow graph. One ALU is allocated for each operator in the program. Because of the self-synchronization of the ALUs, no explicit control or scheduling is needed. Therefore processing these assignments does not access or alter START. The data dependences (as they would be exposed in the DAG representation of the program [1]) are analyzed through the processing of VARLIST. These assignments synchronize themselves through the data-flow. The data-driven execution automatically exploits the available instruction level parallelism.

All assignments evaluate the right-hand side (RHS) or source expression. This evaluation results in a pointer to a CDFG object's output (or pseudo-object as defined below). For integer assignments, the left-hand side (LHS) variable or destination is combined with the RHS result object to form a new pair {LHS, result(RHS)} which is added to the front of VARLIST.

The simplest statement is a constant assigned to an integer:[7]

```
a = 5;
```

It doesn't change the CDFG, but adds {a, 5} to the front of VARLIST. The constant 5 is a "pseudo-object" which only holds the value, but does not refer to a CDFG object. Now VARDEF(a) equals 5 at subseqent program points before a is redefined.

Integer assignments can also combine variables already defined and constants:

```
b = a * 2 + 3;
```

In the AST, the RHS is already converted to an expression tree. This tree is transformed to a combination of old and new CDFG objects (which are added to the CDFG) as follows: Each operator (internal node) of the tree is substituted by an ALU with the opcode corresponding to the operator in the tree. If a leaf node is a constant, the ALU's input is directly connected to that constant. If a leaf note is an integer variable var, it is looked up in VARLIST, i. e. VARDEF(var) is retrieved. Then VARDEF(var) (an output of an already existing object in CDFG or a constant) is connected to the ALU's input. The output of the ALU corresponding to the root operator in the expression tree is defined as the *result* of the RHS. Finally, a new pair {LHS, result(RHS)} is added to VARLIST. If the two assignments above are processed, the

[6] This method of using a VARLIST is adapted from the Transmogrifier C compiler [5].

[7] Note that we use C syntax for the following examples.

CDFG with two ALUs in Fig. 2 is created.[8] Outputs occurring in VARLIST are labeled by Roman numbers. After these two assignments, VARLIST = [{b, I}, {a, 5}]. (The front of the list is on the left side.) Note that all inputs connected to a constant (whether direct from the expression tree or retrieved from VARLIST) must be defined as constant. Inputs defined as constants have a small c next to the input arrow in Fig. 2.

4.2.2 Conditional Integer Assignments

For conditional if-then-else statements containing only integer assignments, objects for condition evaluation are created first. The object event output indictating the condition result is kept for choosing the correct branch result later. Next, both branches are processed in parallel, using separate copies VARLIST1 and VARLIST2 of VARLIST. (VARLIST itself is not changed.) Finally, for all variables added to VARLIST1 *or* VARLIST2, a new entry for VARLIST is created (combination phase). The valid definitions from VARLIST1 and VARLIST2 are combined with a MUX function, and the correct input is selected by the condition result. For variables only defined in one of the two branches, the multiplexer uses the result retrieved from the original VARLIST for the other branch. If the original VARLIST does not have an entry for this variable, a special "undefined" constant value is used. However, in a functionally correct program this value will never be used. As an optimization, only variables *live* [1] after the if-then-else structure need to be added to VARLIST in the combination phase.[9]

Consider the following example:

```
i = 7;
a = 3;
if (i < 10) {
  a = 5;
  c = 7;
}
else {
  c = a - 1;
  d = 0;
}
```

Fig. 3 shows the resulting CDFG. Before the if-then-else construct, VARLIST = [{a, 3}, {i, 7}]. After processing the branches, for the then branch, VARLIST1 = [{c, 7}, {a, 5}, {a, 3}, {i, 7}], and for the else branch, VARLIST2 = [{d, 0}, {c, I}, {a, 3}, {i, 7}]. After combination, VARLIST = [{d, II}, {c, III}, {a, IV}, {a, 3}, {i, 7}].

Note that case- or switch-statements can be processed, too, since they can – without loss of generality – be converted to nested if-then-else statements.

Processing conditional statements this way does not require explicit control and does not change START. Both branches are executed in parallel and synchronized by the data-flow. It is possible to pipeline the dataflow for optimal throughput.

[8] Note that the input and output names can be deduced from their position, cf. Fig. 1. Also note that the compiler frontend would normally have substituted the second assignment by b = 13 (constant propagation). For the simplicity of this explanation, no frontend optimizations are considered in this and the following examples.

[9] Definition: A variable is *live* at a program point if its value is read at a statement reachable from here without intermediate redefinition.

4.2.3 General Conditional Statements

Conditional statements containing either array accesses (cf. Section 4.2.7 below) or inner loops cannot be processed as described in Section 4.2.2. Data packets must only be sent to the active branch. This is achieved by the implementation shown in Fig. 8, similar to the method presented in [4].

A dataflow analysis is performed to compute *used sets* `use` and *defined sets* `def` [1] of both branches.[10] For the current VARLIST entries of all variables in $IN = use(thenbody) \cup def(thenbody) \cup use(elsebody) \cup def(elsebody) \cup use(header)$, DEMUX functions controlled by the IF condition are inserted. Note that arrows with double lines in Fig. 8 denote connections for all variables in IN, and the shaded DEMUX function stands for several DEMUX functions, one for each variable in IN. The DEMUX functions forward data packets only to the selected branch. New lists VARLIST1 and VARLIST2 are compiled with the respective outputs of these DEMUX functions. The then-branch is processed with VARLIST1, and the else branch with VARLIST2. Finally, the output values are combined. OUT contains the new values for the same variables as in IN. Since only one branch is ever activated there will not be a conflict due to two packets arriving simultanuously. The combinations will be added to VARLIST after the conditional statement. If the IF execution shall be pipelined, MERGE opcodes for the output must be inserted, too. They are controlled by the condition like the DEMUX functions.

The following extension with respect to [4] is added (dotted lines in Fig. 8) in order to control the execution as mentioned above with START events: The START input is ECOMB-combined with the condition output and connected to the SEL input of the DEMUX functions. The START inputs of thenbody and elsebody are generated from the ECOMB output sent through a 1-FILTER and a 0-CONSTANT[11] or through a 0-FILTER, respectively. The overall $START_{new}$ output is generated by a simple "2 to 1 connection" of thenbody's and elsebody's $START_{new}$ outputs. With this extension, arbitrarily nested conditional statements or loops can be handled within thenbody and elsebody.

4.2.4 WHILE Loops

WHILE loops are processed similarly to the scheme presented in [4], cf. Fig. 9. As in Section 4.2.3, double line connections and shaded MERGE and DEMUX functions represent duplication for all variables in IN. Here $IN = use(whilebody) \cup def(whilebody) \cup use(header)$. The WHILE loop executes as follws: In the first loop iteration, the MERGE functions select all input values from VARLIST at loop entry (SEL=0). The MERGE outputs are connected to the header and the DEMUX functions. If the while condition is true (SEL=1), the input values are forwarded to the whilebody, otherwise to OUT. The output values of the while body are fed back to whilebody's input via the MERGE and DEMUX operators as long as the condition is true. Finally, after the last iteration, they are forwarded to OUT. The outputs are added to the new VARLIST.[12]

Two extensions with respect to [4] are added (dotted lines in Fig. 9):

[10] A variable is *used* in a statement (and hence in a program region containing this statement) if its value is read. A variable is *defined* in a statement (or region) if a new value is assigned to it.

[11] The 0-CONSTANT is required since START events must always be 0-events.

[12] Note that the MERGE function for variables not live at the loop's beginning and the whilebody's beginning can be removed since its output is not used. For these variables, only the DEMUX function to output the final value is required. Also note that the MERGE functions can be replaced by simple "2 to 1 connections" if the configuration process guarantees that packets from IN1 always arrive at the DEMUX's input before feedback values arrive.

- In [4], the SEL-input of the MERGE functions is preloaded with 0. Hence the loop execution begins immediately and can be executed only once. Instead, we connect the START input to the MERGE's SEL input ("2 to 1 connection" with the header output). This allows to control the time of the start of the loop execution and to restart it.
- The whilebody's START input is connected to the header output, sent through a 1-FILTER/0-CONSTANT combination as above (generates a 0-event for each loop iteration). By ECOMB-combining whilebody's $START_{new}$ output with the header output for the MERGE functions' SEL inputs, the next loop iteration is only started after the previous one has finished. The while loop's $START_{new}$ output is generated by filtering the header output for a 0-event.

With these extensions, arbitrarily nested conditional statements or loops can be handled within whilebody.

4.2.5 FOR Loops

FOR loops are particularly regular WHILE loops. Therefore we could handle them as explained above. However, our RDFP features the special counter function CNT and the data packet multiplication function MDATA which can be used for a more efficient implementation of FOR loops. This new FOR loop scheme is shown in Fig. 10.

A FOR loop is controlled by a counter CNT. The lower bound (LB), upper bound (UB), and increment (INC) expressions are evaluated like any other expressions (see Sections 4.2.1 and 4.2.7) and connected to the respective inputs.

As opposed to WHILE loops, a MERGE/DEMUX combination is only required for variables in $IN1 = def(forbody)$, i. e. those *defined* in forbody.[13] IN1 does not contain variables which are only *used* in forbody, LB, UB, or INC, and does also not contain the loop index variable. Variables in IN1 are processed as in WHILE loops, but the MERGE and DEMUX functions' SEL input is connected to CNT's W output. (The W output does the inverse of a WHILE loop's header output; it outputs a 1-event after the counter has terminated. Therefore the inputs of the MERGE functions and the outputs of the DEMUX functions are swapped here, and the MERGE functions' SEL inputs are preloaded with 1-events.)

CNT's X output provides the current value of the loop index variable. If the final index value is required (live) after the FOR loop, it is selected with a DEMUX function controlled by CNT's U event output (which produces one event for every loop iteration).

Variables in $IN2 = use(forbody) \setminus def(forbody)$, i. e. those defined outside the loop and only used (but not redefined) inside the loop are handled differently. Unless it is a constant value, the variable's input value (from VARLIST) must be reproduced in each loop iteration since it is consumed in each iteration. Otherwise the loop would stall from the second iteration onwards. The packets are reproduced by MDATA functions, with the SEL inputs connected to CNT's U output. The SEL inputs must be preloaded with a 1-event to select the first input. The 1-event provided by the last iteration selects a new value for the next execution of the entire loop.

[13] Note that the MERGE functions can be replaced by simple "2 to 1 connections" as for WHILE loops if the configuration process guarantees that packets from IN1 always arrive at the DEMUX's input before feedback values arrive.

The following control events (dotted lines in Fig. 10) are similar to the WHILE loop extensions, but simpler. CNT's START input is connected to the loop's overall START signal. $START_{new}$ is generated from CNT's W output, sent through a 1-FILTER and 0-CONSTANT. CNT's V output produces one 0-event for each loop iteration and is therefore used as forbody's START. Finally, CNT's NEXT input is connected to forbody's $START_{new}$ output.

For pipelined loops (as defined below in Section 4.2.6), loop iterations are allowed to overlap. Therefore CNT's NEXT input needs not be connected. Now the counter produces index variable values and control events as fast as they can be consumed. However, in this case CNT's W output in not sufficient as overall $START_{new}$ output since the counter terminates before the last iteration's forbody finishes. Instead, $START_{new}$ is generated from CNT's U output ECOMB-combined with forbody's $START_{new}$ output, sent through a 1-FILTER/0-CONSTANT combination. The ECOMB produces an event after termination of each loop iteration, but only the *last* event is a 1-event because only the last output of CNT's U output is a 1-event. Hence this event indicates that the last iteration has finished. Cf. Section 4.3 for a FOR loop example compilation with and without pipelining.

As for WHILE loops, these methods allow to process arbitrarily nested loops and conditional statements. The following advantages over WHILE loop implementations are achieved:

- One index variable value is generated by the CNT function each clock cycle. This is faster and smaller than the WHILE loop implementation which allocates a MERGE/DEMUX/ADD loop and a comparator for the counter functionality.
- Variables in IN2 (only used in forbody) are reproduced in the special MDATA functions and need not go through a MERGE/DEMUX loop. This is again faster and smaller than the WHILE loop implementation.

4.2.6 Vectorization and Pipelining

The method described so far generates CDFGs performing the HLL program's functionality on an RDFP. However, the program execution is unduly sequentialized by the START signals. In some cases, innermost loops can be *vectorized*. This means that loop iterations can overlap, leading to a pipelined dataflow through the operators of the loop body. The *Pipeline Vectorization* technique [6] can be easily applied to the compilation method presented here. As mentioned above, for FOR loops, the CNT's NEXT input is removed so that CNT counts continuously, thereby overlapping the loop iterations.

All loops without array accesses can be pipelined since the dataflow automatically synchronizes *loop-carried dependences*, i. e. dependences between a statement in one iteration and another statement in a subsequent iteration. Loops with array accesses can be pipelined if the array (i. e. RAM) accesses do not cause loop-carried dependences or can be transformed to such a form. In this case no RAM address is written in one and read in a subsequent iteration. Therefore the read and write accesses to the same RAM may overlap. This degree of freedom is exploited in the RAM access technique described below. Especially for dual-ported RAM it leads to considerable performance improvements.

4.2.7 Array Accesses

In contrast to scalar variables, array accesses have to be controlled explicitly in order to maintain the program's correct execution order. As opposed to normal dataflow machine models [3], a RDFP does

not have a single address space. Instead, the arrays are allocated to several RAMs. This leads to a different approach to handling RAM accesses and opens up new opportunities for optimization.

To reduce the complexity of the compilation process, array accesses are processed in two phases. Phase 1 uses "pseudo-functions" for RAM read and write accesses. A RAM read function has a RD data input (read address) and an OUT data output (read value), and a RAM write function has WR and IN data inputs (write address and write value). Both functions are labeled with the array the access refers to, and both have a START event input and a U event output. The events control the access order. In Phase 2 all accesses to the same RAM are combined and substituted by a single RAM function as shown in Fig. 1. This involves manipulating the data and event inputs and outputs such that the correct execution order is maintained and the outputs are forwarded to the correct part of the CDFG.

Phase 1 Since arrays are allocated to several RAMs, only accesses to the same RAM have to be synchronized. Accesses to different RAMs can occur concurrently or even out of order. In case of data dependencies, the accesses self-synchronize automatically. Within pipelined loops, not even read and write accesses to the same RAM have to be synchronized. This is achieved by maintaining separate START signals for every RAM or even separate START signals for RAM read and RAM write accesses in pipelined loops. At the end of a basic block [1][14], all $START_{new}$ outputs must be combined by a ECOMB to provide a START signal for the next basic block which guarantees that all array accesses in the previous basic block are completed. For pipelined loops, this condition can even be relaxed. Only after the loop exit all accesses have to be completed. The individual loop iterations need not be synchronized.

First the RAM addresses are computed. The compiler frontend's standard transformation for array accesses can be used, and a CDFG function's output is generated which provides the address. If applicable, the offset with respect to the RDFP RAM (as determined in the initial mapping phase) must be added. This output is connected to the pseudo RAM read's RD input (for a read access) or to the pseudo RAM write's WR input (for a write access). Additionally, the OUT output (read) or IN input (write) is connected. The START input is connected to the variable's START signal, and the U output is used as $START_{new}$ for the next access.

To avoid redundant read accesses, RAM reads are also registered in VARLIST. Instead of an integer variable, an array element is used as first element of the pair. However, a change in a variable occurring in an array index invalidates the information in VARLIST. It must then be removed from it.

The following example with two read accesses compiles to the intermediate CDFG shown in Fig. 12. The START signals refer only to variable a. STOP1 is the event connection which synchronizes the accesses. Inputs START (old), i and j should be substituted by the actual outputs resulting from the program before the array reads.

```
x = a[i];
y = a[j];
z = x + y;
```

Fig. 13 shows the translation of the following write access:

```
a[i] = x;
```

[14] A *basic block* is a program part with a single entry and a single exit point, i.e. a piece of straight-line code.

Phase 2 We now merge the pseudo-functions of all accesses to the same RAM and substitute them by a single RAM function. For all data inputs (RD for read access and WR and IN for write access), GATEs are inserted between the input and the RAM function. Their E inputs are connected to the respective START inputs of the original pseudo-functions. If a RAM is read and written at only one program point, the U output of the read and write access is moved to the ERD or EWR output, respectively. For example, the single access `a[i] = x;` from Fig. 13 is transformed to the final CDFG shown in Fig. 5.

However, if several read or several write accesses (i. e. pseudo-functions from different program points) to the same RAM occur, the ERD or EWR events are not specific anymore. But a $START_{new}$ event of the original pseudo function should only be generated for the respective program point, i. e. for the *current access*. This is achieved by connecting the START signals of all *other* accesses (pseudo-functions) of the same type (read or write) with the *inverted* START signal of the current access. The resulting signal produces an event for every access, but only for the current access a 1-event. This event is ECOMB-combined with the RAM's ERD or EWR output. The ECOMB's output will only occur after the access is completed. Because ECOMB OR-combines its event packets, only the current access produces a 1-event. Next, this event is filtered with a 1-FILTER and changed by a 0-CONSTANT, resulting in a $START_{new}$ signal which produces a 0-event only after the current access is completed as required.

For several accesses, several sources are connected to the RD, WR and IN inputs of a RAM. This disables the self-synchronization. However, since only one access occurs at a time, the GATEs only allow one data packet to arrive at the inputs.

For read accesses, the packets at the OUT output face the same problem as the ERD event packets: They occur for every read access, but must only be used (and forwarded to subsequent operators) for the current access. This can be achieved by connecting the OUT output via a DEMUX function. The Y output of the DEMUX is used, and the X output is left unconnected. Then it acts as a selective gate which only forwards packets if its SEL input receives a 1-event, and discards its data input if SEL receives a 0-event. The signal created by the ECOMB described above for the $START_{new}$ signal creates a 1-event for the current access, and a 0-event otherwise. Using it as the SEL input achieves exactly the desired functionality.

Fig. 4 shows the resulting CDFG for the first example above (two read accesses), after applying the transformations of Phase 2 to Fig. 12. STOP1 is now generated as follws: START(old) is inverted, "2 to 1 connected" to STOP1 (because it is the START input of the second read pseudo-function), ECOMB-combined with RAM's ERD output and sent through the 1-FILTER/0-CONSTANT combination. START(new)is generated similarly, but here START(old) is directly used and STOP1 inverted. The GATEs for input IN (i and j) are connected to START(old) and STOP1, respectively, and the DEMUX functions for outputs x and y are connected to the ECOMB outputs related to STOP1 and START(new).

Multiple write accesses use the same control events, but instead of one GATE per access for the RD inputs, one GATE for WR and one gate for IN (with the same E input) are used. The EWR output is processed like the ERD output for read accesses.

This transformation ensures that all RAM accesses are executed correctly, but it is not very fast since read or write accesses to the same RAM are not pipelined. The next access only starts after the previous one is completed, even if the RAM being used has several pipeline stages. This inefficiency can be removed as follws:

First continuous *sequences* of either read accesses or write accesses (not mixed) within a basic block are detected by checking for pseudo-functions whose U output is directly connected to the START input of another pseudo-function of the same RAM and the same type (read or write). For these sequences, it is

possible to stream data into the RAM rather than waiting for the previous access to complete. For this purpose, a combination of MERGE functions selects the RD or WR and IN inputs in the order given by the sequence. The MERGEs must be controlled by iterative ESEQs guaranteeing that the inputs are only forwarded in the desired order. Then only the first access in the sequence needs to be controlled by a GATE or GATEs. Similarly, the OUT outputs of a read access can be distributed more efficiently for a sequence. A combination of DEMUX functions with the same ESEQ control can be used. It is most efficient to arrange the MERGE and DEMUX functions as balanced binary trees.

The $START_{new}$ signal is generated as follows: For a sequence of length n, the START signal of the entire sequence is replicated n times by an ESEQ[00..1] function with the START input connected to the sequence's START. Its output is directly "N to 1 connected" with the other accesses' START signal (for single accesses) or ESEQ outputs sent through 0-CONSTANT (for access sequences), ECOMB-connected to EWR or ERD, respectively, and sent through a 1-FILTER/0-CONSTANT combination, similar to the basic method described above. Since only the last ESEQ output is a 1-event, only the *last* RAM access generates a $START_{new}$ as required. Alternatively, for read accesses, the generation of the last output can be sent through a GATE (without the E input connected), thereby producing a $START_{new}$ event.

Fig. 14 shows the optimized version of the first example (Figures 12 and 4) using the ESEQ-method for generating $START_{new}$, and Fig. 6 shows the final CDFG of the following, larger example with three array reads. Here the latter method for producing the $START_{new}$ event is used.

```
x = a[i];
y = a[j];
z = a[k];
```

If several read sequences or read sequences and single read accesses occur for the same RAM, 1-events for detecting the *current accesses* must be generated for sequences of read accesses. They are needed to separate the OUT-values relating to separate sequences. The ESEQ output just defined, sent through a 1-CONSTANT, achieves this. It is again "N to 1 connected" to the other accesses' START signals (for single accesses) or ESEQ outputs sent through 0-CONSTANT (for access sequences). The resulting event is used to control a first-stage DEMUX which is inserted to select the relevant OUT output data packets of the sequence as described above for the basic method. Refer to the second example (Figures 15 and 16) in Section 4.3 for a complete example.

4.2.8 Input and Output Ports

Input and output ports are processed similar to vector accesses. A read from an input port is like an array read without an address. The input data packet is sent to DEMUX functions which send it to the correct subsequent operators. The STOP signal is generated in the same way as described above for RAM accesses by combining the INPORT's U output with the current and other START signals.

Output ports control the data packets by GATEs like array write accesses. The STOP signal is also created as for RAM accesses.

4.3 More Examples

Fig. 7 shows the generated CDFG for the following for loop.

```
a = b + c;
for (i=0; i<=10; i++) {
  a = a + i;
  x[i] = k;
}
```

In this example, $IN1 = \{a\}$ and $IN2 = \{k\}$ (cf. Fig. 10). The MERGE function for variable a is replaced by a 2:1 data connection as mentioned in the footnote of Section 4.2.5. Note that only one data packet arrives for variables b, c and k, and one final packet is produced for a (out). forbody does not use a START event since both operations (the adder and the RAM write) are dataflow-controlled by the counter anyway. But the RAM's EWR output is the forbody's $START_{new}$ and connected to CNT's NEXT input. Note that the pipelining optimization, cf. Section 4.2.6, was not applied here. If it is applied (which is possible for this loop), CNT's NEXT input is not connected, cf. Fig. 11. Here, the loop iterations overlap. $START_{new}$ is generated from CNT's U output and forbody's $START_{new}$ (i.e. RAM's EWR output), as defined at the end of Section 4.2.5.

The following program contains a vectorizable (pipelined) loop with one write access to array (RAM) x and a sequence of two read accesses to array (RAM) y. After the loop, another single read access to y occurs.

```
z = 0;
for (i=0; i<=10; i++) {
  x[i] = i;
  z = z + y[i] + y[2*i];
}
a = y[k];
```

Fig. 15 shows the intermediate CDFG generated before the array access Phase 2 transformation is applied. The pipelined loop is controlled as follows: Within the loop, separate START signals for write accesses to x and read accesses to y are used. The reentry to the forbody is also controlled by two independent signals ("cycle1" and "cycle2"). For the read accesses, "cycle2" guarantees that the read y accesses occur in the correct order. But the beginning of an iteration for read y and write x accesses is not synchronized. Only at loop exit all accesses must be finished, which is guaranteed by signal "loop finished". The single read access is completely independent of the loop.

Fig. 16 shows the final CDFG after Phase 2. Note that "cycle1" is removed since a single write access needs no additional control, and "cycle2" is removed since the inserted MERGE and DEMUX functions automatically guarantee the correct execution order. The read y accesses are not independent anymore since they all refer to the same RAM, and the functions have been merged. ESEQs have been allocated to control the MERGE and DEMUX functions of the read sequence, and for the first-stage DEMUX functions which separate the read OUT values for the read sequence and for the final single read access. The ECOMBs, 1-FILTERs, 0-CONSTANTs and 1-CONSTANTs are allocated as described in Section 4.2.7, Phase 2, to generate correct control events for the GATEs and DEMUX functions.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] The Stanford SUIF Compiler Group. Homepage `http:// suif.stanford.edu`.

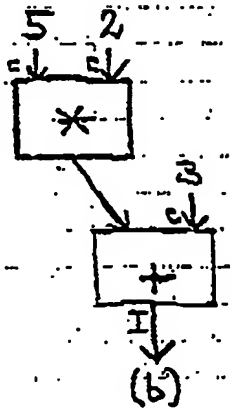
[3] A. H. Veen. Dataflow architecture. *ACM Computing Surveys*, 18(4), December 1986.

[4] S. J. Allan and A. E. Oldehoeft. A flow analysis procedure for the translation of high-level languages to a data flow language. *IEEE Transactions on Computers*, C-29(9):826–831, September 1980.

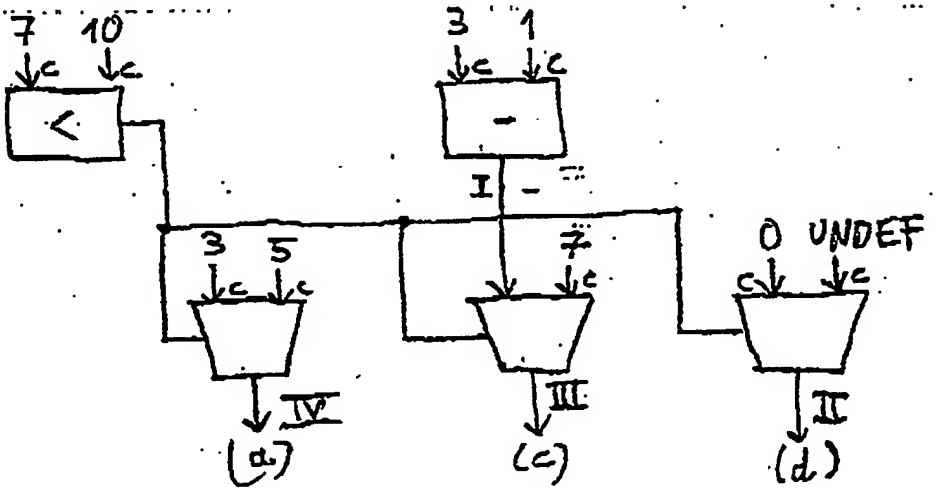
[5] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In *Proc. FPGAs for Custom Computing Machines*, pages 136–144. IEEE Computer Society Press, 1995.

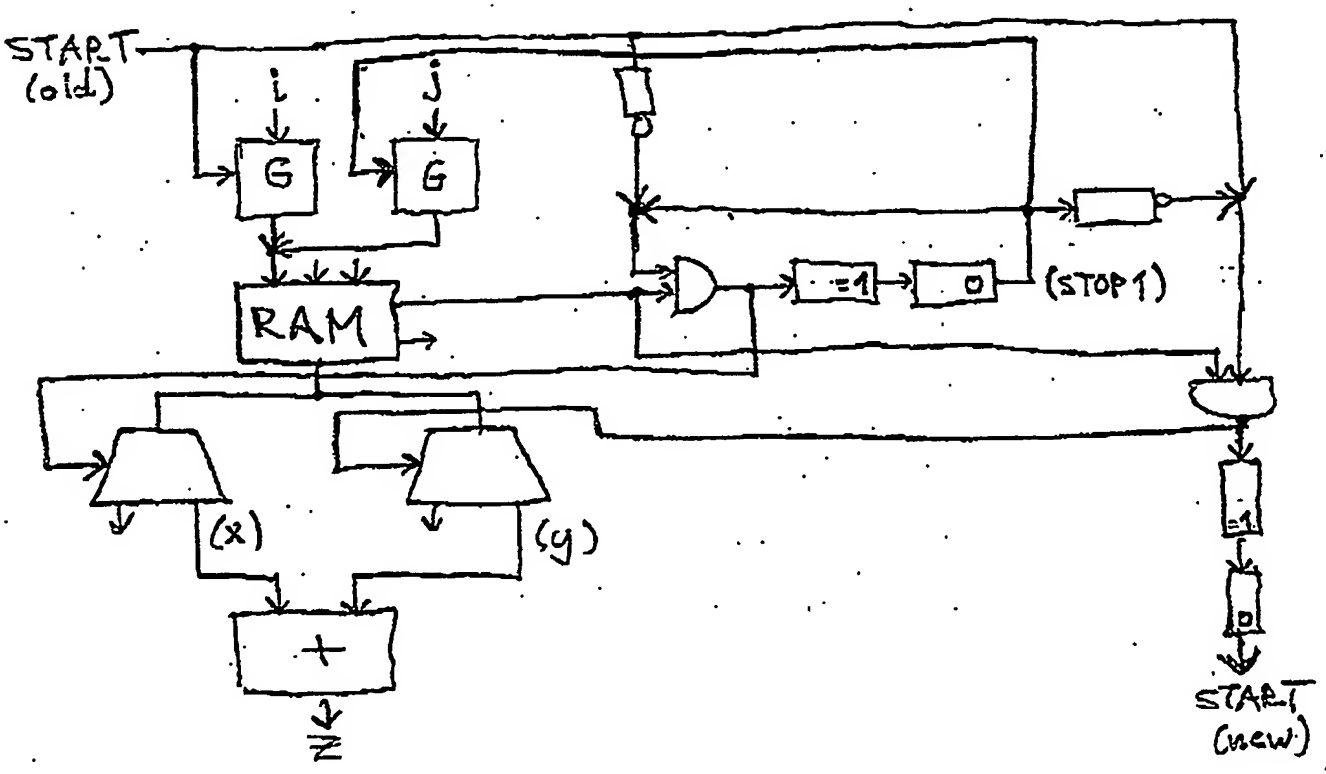
[6] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2), February 2001.

2:



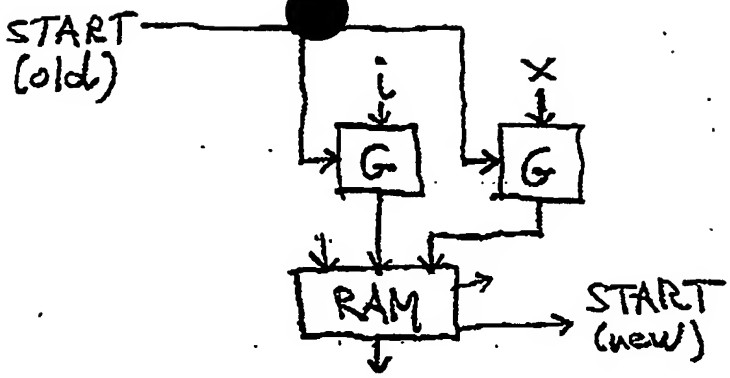
3:



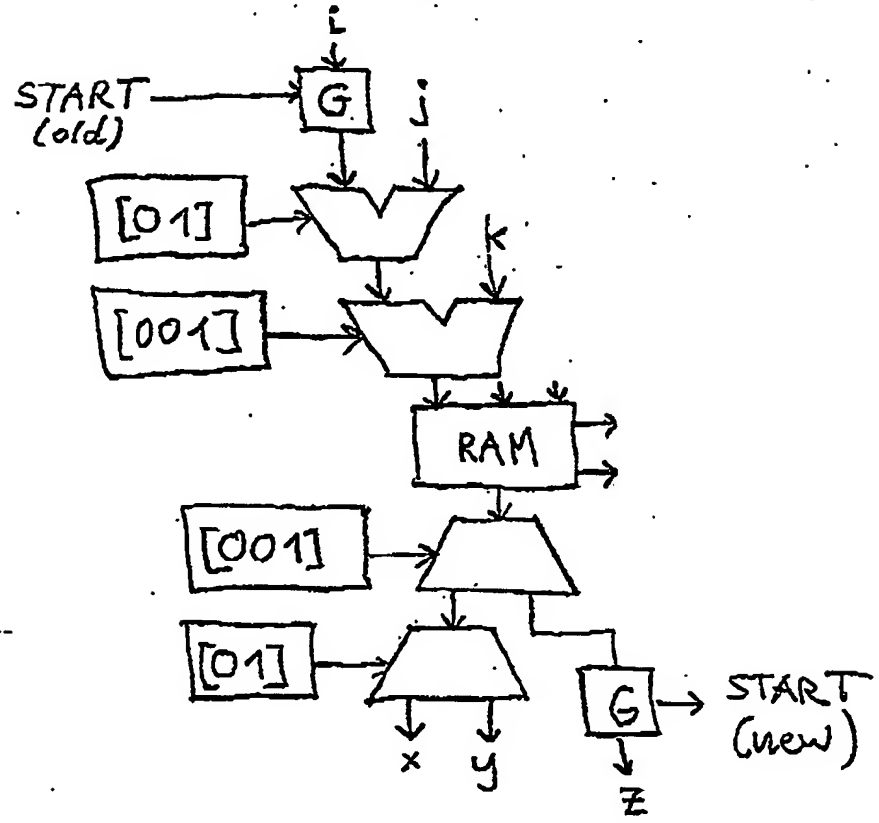


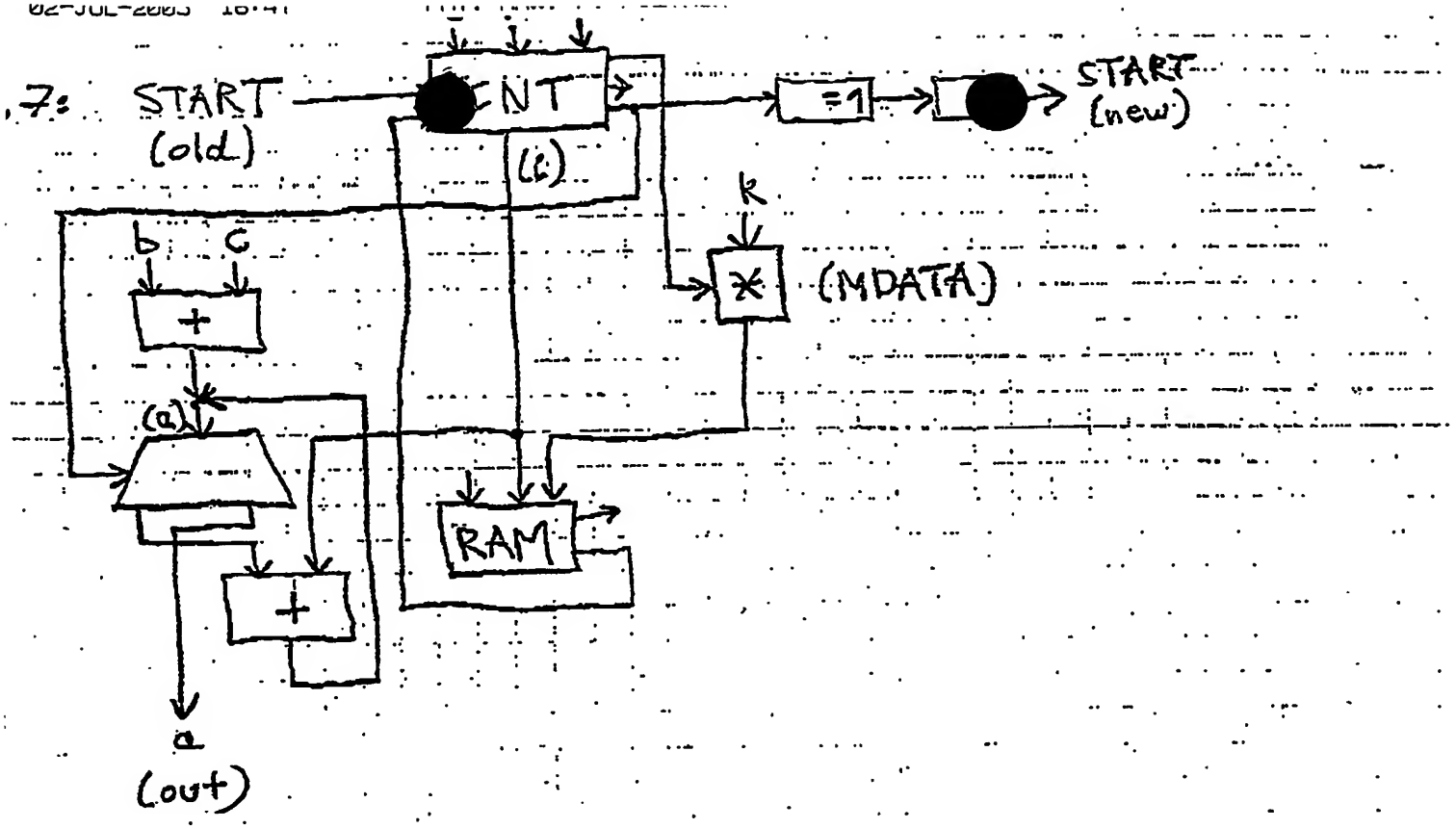
(Figures: 2 pages)

1.5:


START
(old)
i
x
G
G
RAM
START
(new)

1.6:

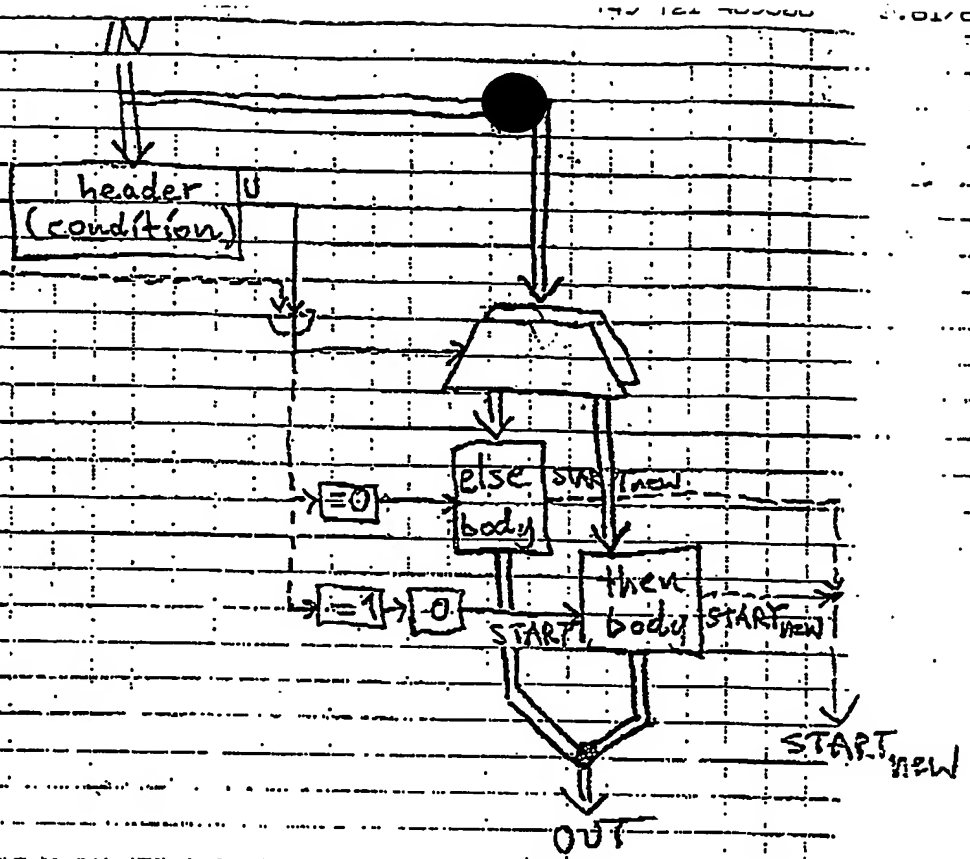

i
START
(old)
G
j
[01]
k
[001]
RAM
[001]
[01]
G
START
(new)
x
y
z


7:
START
(old)
NT
(i)
=1
START
(new)
k
b
c
+
*
(MDATA)
(a)
RAM
+
a
(out)

8:

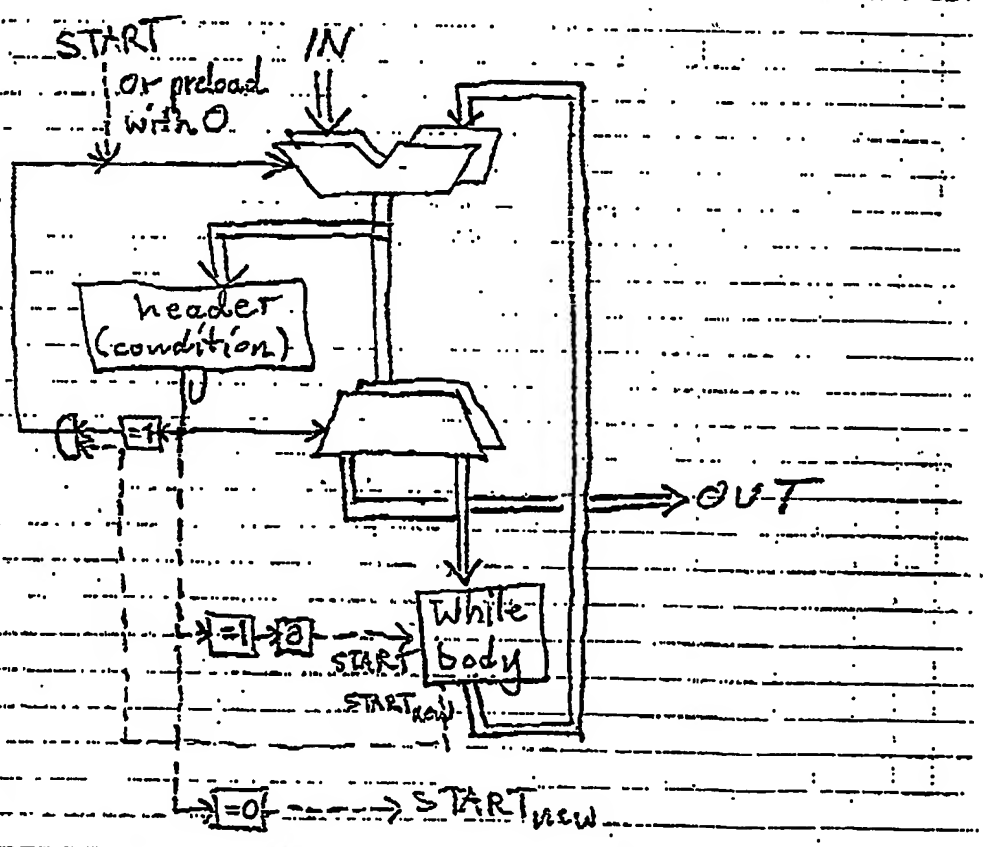
eral Conditional

temcut Template



9:

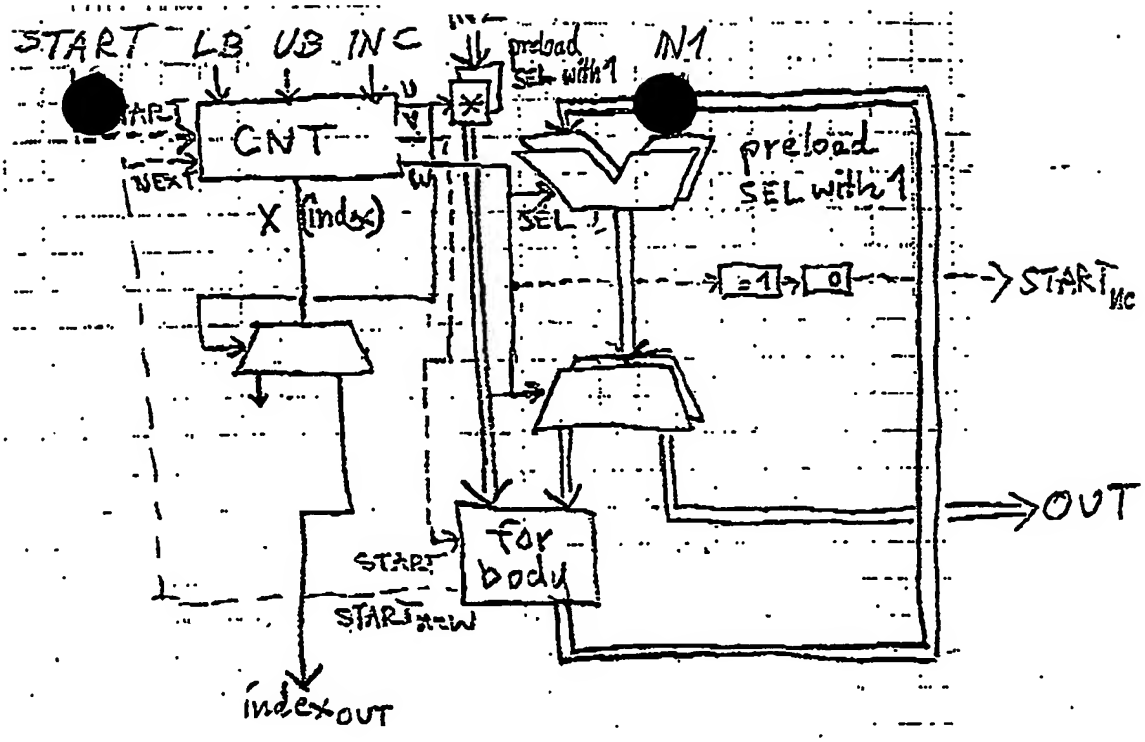
e Loop Template



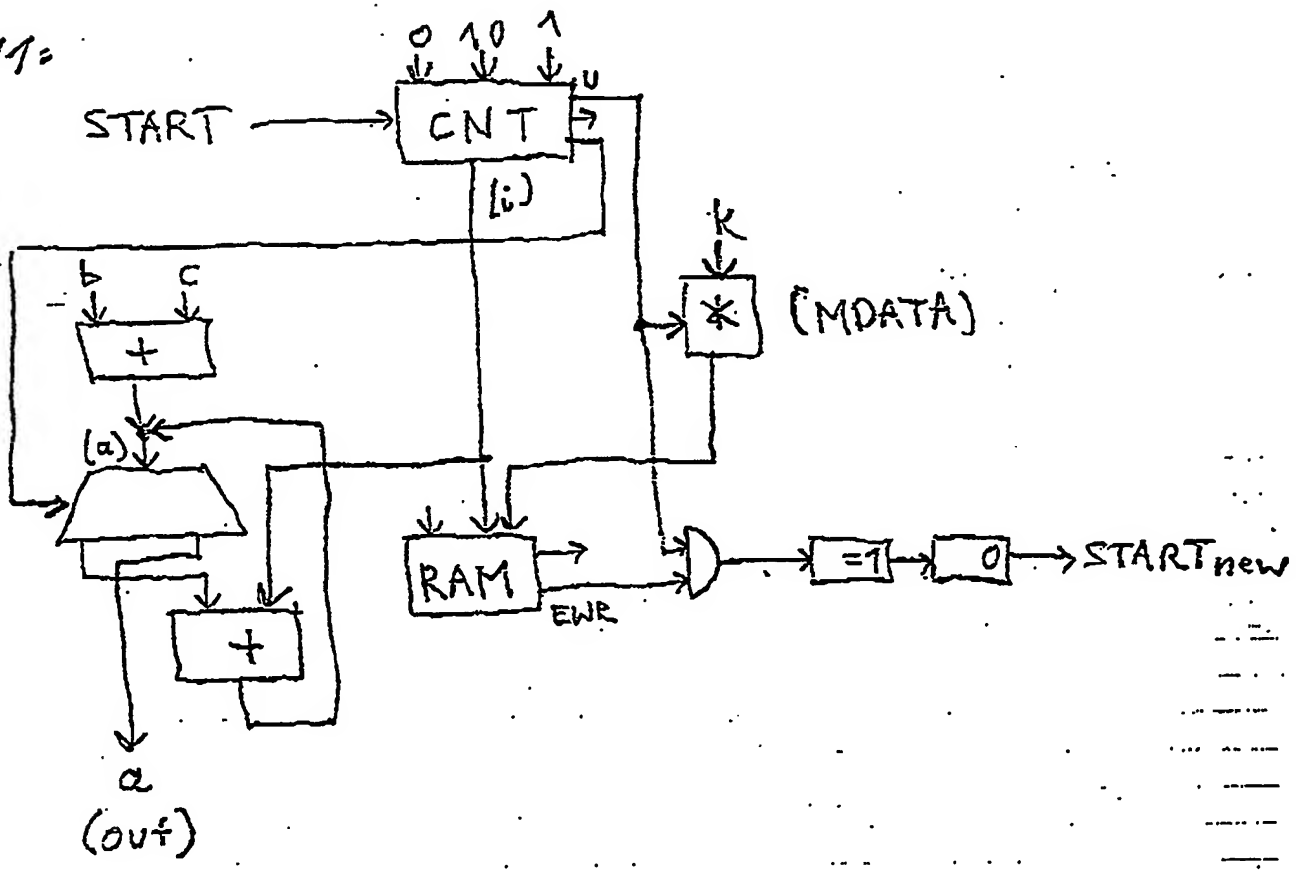
02-JUL-2003 16:40

10:

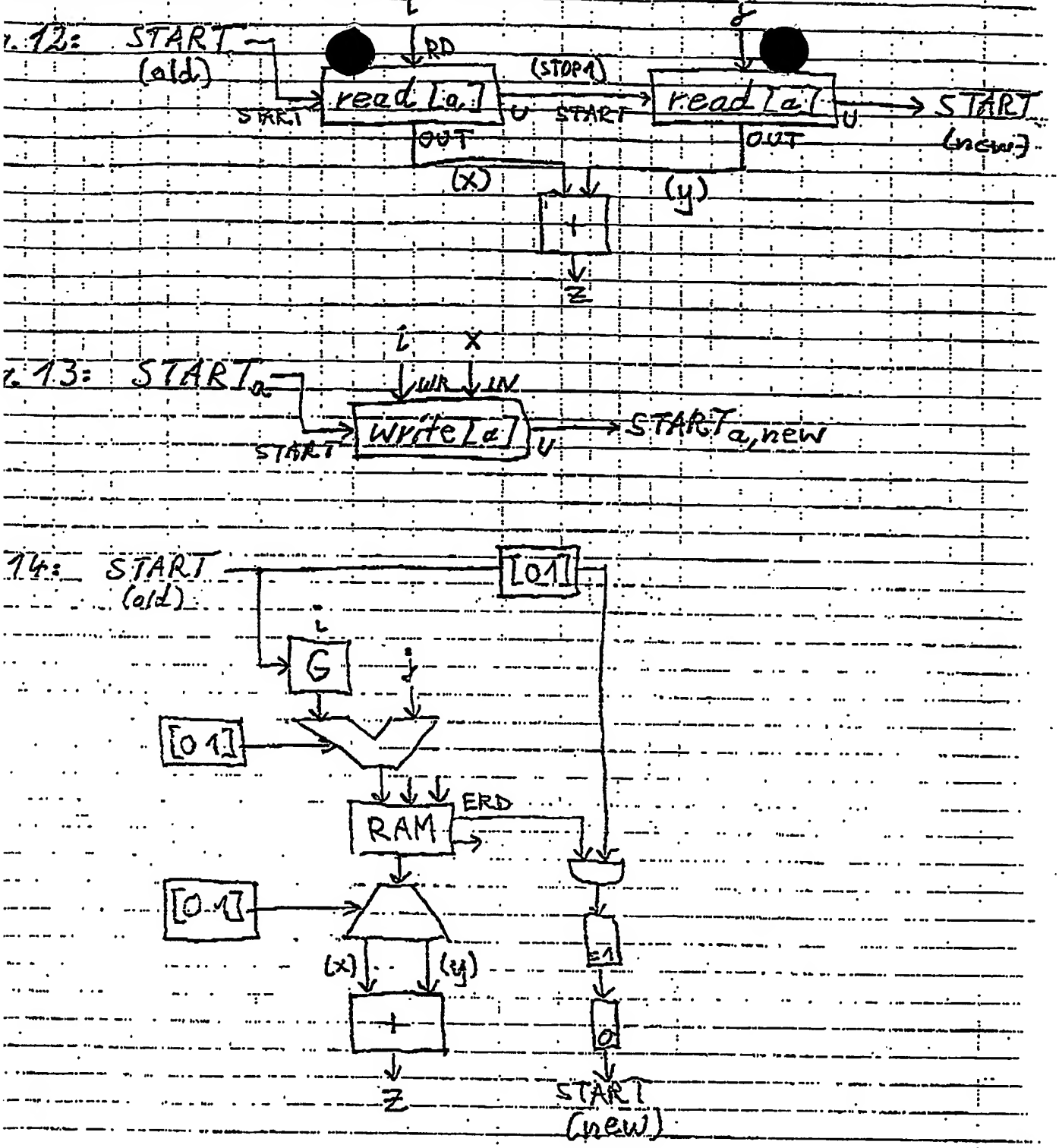
Loop Template

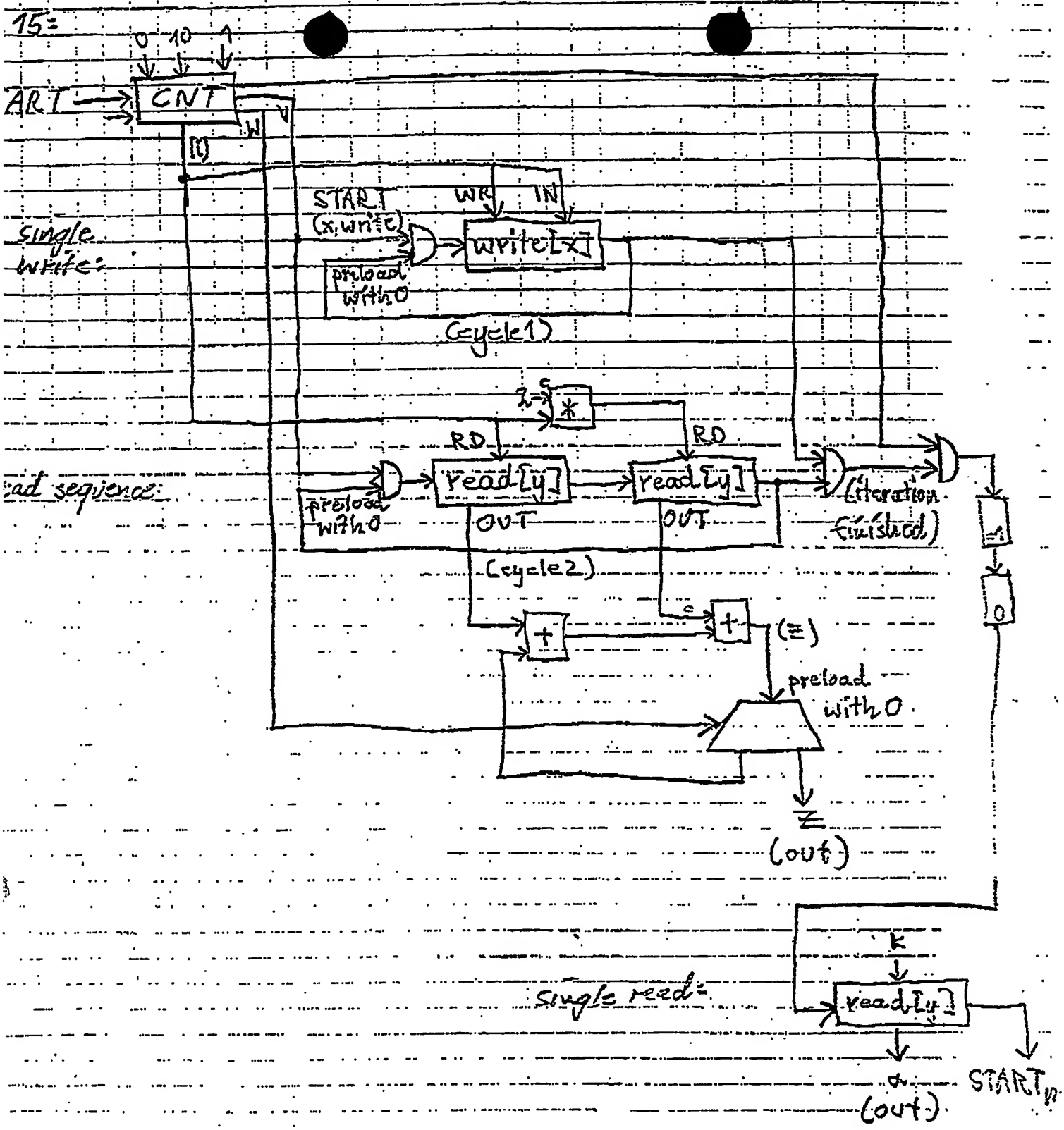


11:

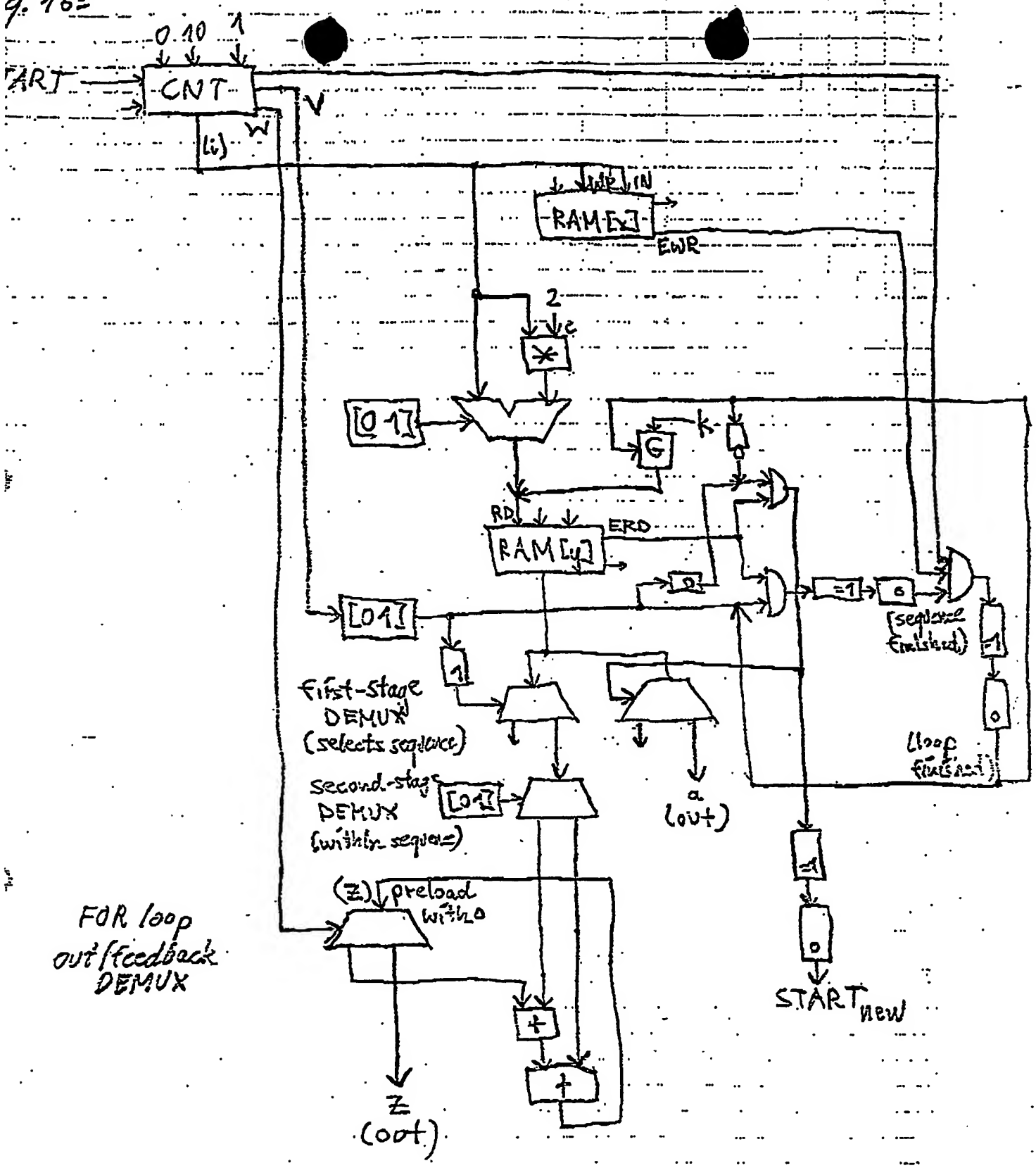


Empfangszeit 2.Juli 16:22


12: START (old)
RD
read [a]
START
OUT
(x)
(STOP1)
read [a]
START
OUT
(y)
START (new)
+
z
13: START a
i
x
WR
IN
write [a]
START
START a, new
14: START (old)
[01]
G
[01]
RAM
ERD
[01]
(x)
(y)
+
z
=1
a
START (new)


15=
0
10
1
ART
CNT
W
IN
START
(x write)
WR
IN
write[x]
single
write:
preload
with 0
(cycle1)
c
*
RD
RD
read[y]
read[y]
preload
with 0
OUT
OUT
(iteration
finished)
(cycle2)
+
+
(=)
preload
with 0
Z
(out)
-1
0
k
single read:
read[y]
a
(out)
START_n

9.16=


0 10 1
START
CNT
V
W
(i)
WR IN
RAM[x]
EWR
2
c
*
[0 1]
G
RD
ERD
RAM[y]
[0 1]
=1
0
[sequence finished]
1
0
First-Stage DEMUX (selects sequence)
Second-stage DEMUX (within sequence)
[0 1]
a (out)
loop finished
(z) preload with 0
FOR loop out feedback DEMUX
+
+
z (out)
START new

Akte: PACT47/EP

European Patent Application

Applicant: PACT XPP Technologies AG
Muthmannstrasse 1
80939 München

Representative: European Patent Attorney
Claus Peter Pietruk
Heinrich-Lilienfein-Weg 5
D-76299 Karlsruhe
Vertreter-Nr. 0 085 850

Title: Method for operating an array of reconfigurable elements processing data

Claim

1. Method for operating a reconfigurable array, wherein the array is coupled to a conventional, preferably a RISC-processor and wherein said coupling of said reconfigurable array to said processor is effected via coupling the array to the memory hierarchy of said processor.

-.-